TOWARDS A VERIFIED COMPLEX PROTOCOL STACK IN A PRODUCTION KERNEL: METHODOLOGY AND DEMONSTRATION

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Peter C. Johnson

DARTMOUTH COLLEGE

Hanover, New Hampshire

May, 2016

Dartmouth Computer Science Technical Report TR2016-803

Examining Committee:

(chair) Sean W. Smith, Ph.D.

(co-chair) Sergey Bratus, Ph.D.

David F. Kotz, Ph.D.

Devin Balkcom, Ph.D.

M. Douglas McIlroy, Ph.D.

Trent Jaeger, Ph.D.

F. Jon Kull, Ph.D.
Dean of Graduate Studies

ABSTRACT

Any useful computer system performs communication and any communication must be parsed before it is computed upon. Given their importance, one might expect parsers to receive a significant share of attention from the security community. This is, however, not the case: bugs in parsers continue to account for a surprising portion of reported and exploited vulnerabilities.

In this thesis, I propose a methodology for supporting the development of software that depends on parsers—such as anything connected to the Internet—to safely support any reasonably designed protocol: data structures to describe protocol messages; validation routines that check that data received from the wire conforms to the rules of the protocol; systems that allow a defender to inject arbitrary, crafted input so as to explore the effectiveness of the parser; and systems that allow for the observation of the parser code while it is being explored.

Then, I describe principled method of producing parsers that automatically generates the myriad parser-related software from a description of the protocol. This has many significant benefits: it makes implementing parsers simpler, easier, and faster; it reduces the trusted computing base to the description of the protocol and the program that compiles the description to runnable code; and it allows for easier formal verification of the generated code.

I demonstrate the merits of the proposed methodology by creating a description of the USB protocol using a domain-specific language (DSL) embedded in Haskell and integrating it with the FreeBSD operating system. Using the industry-standard umap test-suite, I measure the performance and efficacy of the generated parser. I show that it is stable, that it is effective at protecting a system from both accidentally and maliciously malformed input, and that it does not incur unreasonable overhead.

Acknowledgements

First and foremost, to my parents and my aunt, without whose support, encouragement, and enthusiasm I wouldn't be here (for an extremely broad definition of "here").

To the Dartmouth Computer Science Department for their patience with me throughout my time as a graduate student, but most especially Sergey Bratus, Sean Smith, and Tom Cormen for their advisorship during a variety of trials, academic and otherwise.

To Doug McIlroy, who was always infectiously eager to discuss my crazy ideas.

To the other members of my thesis committee—Dave Kotz, Devin Balkcom, and Trent Jaeger—for their responsiveness and input into my research and this document.

To the staff of Dick's House, who provided all manner of care.

To the staff of EBAs, who provided fuel and friendship to keep me working.

To the crowd in North Carolina, whose support I could feel from a thousand miles away.

Finally, the material in this thesis is based in part on work supported by the Department of Energy under Award Numbers DE-OE0000780 and DEOE0000097. The views expressed in this paper are those of the author, and do not necessarily reflect the views of the sponsors.

Contents

1 Introduction **1**

1.1 Ubiquity of Ad-Hoc Protocol Parsing 1

1.2 A Better Way 3

1.3 Automagic Generation 6

1.4 Formal Verification & Complete Mediation 9

1.5 Practical Matters 11

1.6 Summary of Contributions 13

2 Overview of USB: Protocol and Vulnerabilities **15**

2.1 The Protocol 15

2.2 USB As a Gateway to the Kernel 17

2.3 Vulnerabilities 17

2.4 Realization 19

3 Injection **20**

3.1 Facedancer 21

3.2 Host-side Software 22

3.3 Reference Emulations 25

3.4 Code 26

3.5 Realization 26

4 Inspection & Instrumentation **27**

4.1 Instrumentation 28

4.2 Experimental Results 32

5 Generation **37**

5.1 Protocol Definition . . . 38
5.2 Generating the Code . . . 42
5.3 User-Defined Policies . . . 45
5.4 Protocol: Assemble! . . . 46
5.5 Operation System Integration . . . 47
5.6 Putting It All Together . . . 51
5.7 Realization . . . 51

6 Evaluation **52**

6.1 Methodology . . . 53
6.2 Complete Mediation . . . 57
6.3 Performance . . . 59
6.4 Effectiveness . . . 61
6.5 Categories . . . 62
6.6 Results . . . 65
6.7 Summary . . . 66

7 Conclusion **67**

7.1 Future Work . . . 68

A CVE Classifications **69**

B Code **103**

B.1 Injection . . . 103
B.2 Inspection . . . 103
B.3 Generation . . . 107

List of Tables

3.1 Analogous features between Ethernet networks—which we have much experience securing—and USB infrastructure. (Reproduced from the paper that presented this work at the Workshop on Embedded Systems Security [8].) 24

4.1 Summary of static probes in our instrumentation framework. 29

5.1 The files, both automagically and manually generated, that comprise the USB validation proof-of-concept; along with their sizes, measured in lines of code, and a brief description of their purpose. 49

6.1 Data sent by umap fuzz-testing. 56

6.2 Complete mediation test results. For each test, shows number of USB frames sent by umap and the number of frames processed by the USB firewall of the machine being tested. 57

6.3 Specifications of machines used for performance testing (Section 6.3). 59

6.4 Results of USB firewall performance tests. Columns show measured duration of fuzz-testing suite for each device class, averaged over three runs, first with the firewall disabled, then with the firewall enabled, and the measured impact of enabling the firewall as a percentage. 60

6.5 Results of running FileBench's `singlestreamread` workload on a USB mass storage device, with the firewall enabled and disabled. 61

6.6 Classification of USB mentions in CVE incident reports into how they might be affected by the USB firewall I created. 63

A.1 Vulnerabilities classified as Unrelated. 69

A.2 Vulnerabilities classified as Unclear. 85

A.3 Vulnerabilities classified as Mitigated By Policy. 88

A.4 Vulnerabilities classified as Mitigated By Pattern. 96
A.5 Vulnerabilities classified as Inherently Averted. 97

B.1 Files comprising the library I wrote to enable emulating various USB devices using the Facedancer. 104
B.2 Applications I wrote to emulate various devices. All build on the libraries listed in Table B.1. 104
B.3 Extent of modifications made to apply instrumentation framework described in Chapter 4. 105
B.4 Scripts used to interact with the instrumentation framework described in Chapter 4 and to process its output. 106
B.5 Hand-written source files for programs that generate the USB firewall. 107
B.6 The generated USB firewall source files. 107
B.7 Hand-written source files that facilitate integration of the USB firewall with FreeBSD. 108

List of Figures

1.1 A parser takes an untyped, opaque bytestream and imposes meaning on it according to the protocol specification (in this example, the Internet Protocol [50]) before handing it off to the kernel for interpretation. 2

1.2 Policies written for the `pf` IP firewall. Note that they include both a description of the data using protocol-specific terminology (i.e., protocol type, port number, IP address) as well as an action (e.g., "pass in") to take if the description matches. . . . 4

1.3 Workflow of a DSL-based parser generator. Given a description of the protocol language written in the domain-specific language, the parser generator produces the parser code. 6

2.1 A request from the host for the first 18 bytes of the device descriptor (id 1, byte 4) and the device's response. 16

3.1 The Facedancer board (version 10). 21

3.2 The USB hierarchy as implemented in the host-side Facedancer stack. This figure describes a single USB device that supports two distinct configurations, the first consisting of two interfaces, each with two endpoints, and the second consisting of a single interface with three endpoints. 22

4.1 Example script, written in `D`, that enables custom DTRACE probes `MY_BB_ENTER` and `MY_BB_RETURN`; prints a message when execution reaches any of those points. 29

4.2 Output from the `bb-trace.d` DTRACE script, showing the basic blocks executed in each function. 31

4.3 Indented basic-block trace. (The underlying data is the same as in Figure 4.2.) . . . 31

4.4 Lines of code exercised per-file during a USB thumbdrive insertion, read, write, and removal. 33

4.5 Diagram generated from basic-block trace showing interactions between different modules within FreeBSD's USB subsystem. Nodes represent source files within the FreeBSD USB stack; each directed edge represents a call from a function in one file (origin node) to a function in another file (destination node). 35

5.1 Given a protocol specification, shown in the middle, we can automagically generate the code required to implement support for that protocol, shown in the leaves. . . . 38
5.2 Specification of a Message within a protocol. A Message is defined by a name (a String), a list of Fields, and a data length specifier. 39
5.3 Specification of a Field within a Message. A Field is defined by a name (a String), a size (in this case, 8 or 16 bits), and an indication of whether its value is variable or fixed 40
5.4 Specification of the length of the data stage of a Message. The length can be zero, a fixed number of bytes, or a number of bytes given in one of the fields of the Message. 40
5.5 Description of USB protocol's `GET_DESCRIPTOR` request message, written in the domain-specific language. 41
5.6 Code to derive the `GET_DESCRIPTOR` response from the associated request message, using the domain-specific language. 42
5.7 Generated C structure for the `GET_DESCRIPTOR` request message. 43
5.8 Generated verification function for the `GET_DESCRIPTOR` request message. (Constant-folding in the compiler will optimize away the tacky addition.) 44
5.9 Generated C accessors for the `GET_DESCRIPTOR` request message. (The duplicate "get" substring is not a typo: the first a verb, the second is part of the noun.) 44
5.10 Generated C function for legibly printing a `GET_DESCRIPTOR` request message. 45
5.11 Example user policies for the USB firewall. 46
5.12 When a USB frame arrives or is sent, the FreeBSD USB stack calls the shim function, `fbsd_hook`, which translates the FreeBSD-formatted USB frame metadata to an OS-agnostic format before passing it along to the generated parser/validator function. The resulting action is cascaded back to the kernel. 50

6.1 Diagram of testing setup. Software (in this case, umap) running on the "testing host" (left) causes the Facedancer to emulate a variety of USB devices when connected to the "target" (right). 53

6.2 Output of umap running in identification mode. (Slightly edited to remove umap banner and long lines.) Of the device classes testable by umap, five are supported in the FreeBSD target: audio control, audio stream, human interface devices (mice and keyboards), printers, mass storage (e.g., thumbdrives), and hubs. 54

6.3 Sample output from umap fuzzing Audio Control devices. The "A" in the final command-line argument causes umap to run "all" tests. The umap banner has been removed and the output has been truncated (full output runs 119 lines and is summarized in Table 6.1). 56

6.4 To capture USB data sent by the Facedancer, I connected the Beagle USB 12 Protocol Analyzer as a pass-through device between the Facedancer and the FreeBSD target; then I connected the Analysis port of the Beagle back to the testing host to capture packets. 58

Chapter 1

Introduction

Protocols are rules by which two or more entities communicate to accomplish some task. In the context of computing, these entities are software or hardware and the tasks may be as mundane as transferring a file between two computers or as complex as many people playing a real-time game. Protocols such as IP [50] achieve the goal of sending arbitrary data across the Internet; USB [25] allows a multitude of different devices to plug into the same physical port on a computer; ELF [16] lets a compiler describe a program in such a way that the kernel can run it; proprietary protocols enable players to shoot virtual rocket launchers at each other from across the globe. In short, to say protocols are ubiquitous would run the risk of underselling them.

The protocols used by modern computers are constantly evolving: new ones are being developed and old ones are being repurposed; it is unrealistic to assume either will ever stop happening. Old protocols are frequently re-implemented (for example, in embedded systems) and introduce new, potentially vulnerable artifacts into the wild. At the same time, new protocols bring their own set of baggage to the security table because with them come new specifications, new implementations, and therefore new vulnerabilities. Engineers have been developing, implementing, and deploying new protocols for decades and yet still we see new vulnerabilities and exploits against them (Chapter 2 contains a survey).

1.1 Ubiquity of Ad-Hoc Protocol Parsing

In all protocols, otherwise-meaningless bits are communicated via some medium and meaning must be applied to them by the receiving entity. In IP, this is the opaque payload carried by the underlying link layer; in USB, these are the typeless bits flying over the physical wire; in ELF, this is the inert

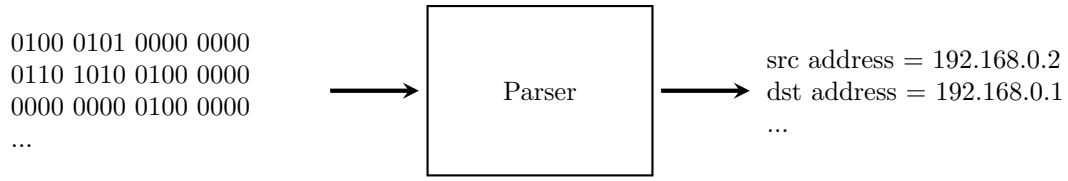

Figure 1.1: A parser takes an untyped, opaque bytestream and imposes meaning on it according to the protocol specification (in this example, the Internet Protocol [50]) before handing it off to the kernel for interpretation.

file sitting on disk. The protocol specification defines both the *syntax* of this communication—the grouping of individual bits within an otherwise-opaque datastream into distinct, typed fields—as well as the *semantics*—how to interpret and respond to the various fields delineated by the syntactic rules. Figure 1.1 shows a stylized representation of the parser's job in, e.g., the Internet Protocol [50].

The component responsible for implementing a protocol syntax is called a *parser* and often sits directly between the medium carrying the opaque data and the code implementing the communication semantics. One might be tempted to dismiss parsers as trivial pieces of software that don't merit much attention. This is far from the case, however. The parser is the key front-end component responsible for deriving meaning from input: for applying semantics to raw bits: for guarding the passage between the untrusted outside world and the vulnerable internals. It's what takes a stream of opaque bits from the ether and decides they comprise a 32-bit unsigned big-endian integer or a stream of null-terminated 8-bit ASCII characters and so on. Put another way, the parser takes arbitrary, untyped data and imposes on it a type before passing that datum on to the rest of the system for further processing.

Frequently, the "rest of the system" implicitly trusts the type that the parser has imposed; the venerable buffer overflow attack exemplifies this trust as well as its fragility. (This perspective, in fact, applies to most memory corruption attacks [40].) One view of such an attack is that the vulnerable function assumed the parser was delivering a null-terminated string of length no more than x when in fact it made no such guarantee. In the case of stack-based buffer overflows, this allows a nefarious (or incompetent) input-provider to overwrite portions of the stack and disrupt execution [45]. Such exploits cannot occur when the characteristics of data guaranteed by the parser match the data-consumer's expectations about it.

While they are near-mythic at this point, stack-based buffer overflows such as described in AlephOne's seminal paper [45] are by no means the only vulnerabilities that would be prevented by

mindful implementation of parsers. In 1998, Thomas Ptacek and Timothy Newsham showed that parsers in TCP/IP stacks of popular operating systems were sufficiently varied and idiosyncratic in their behavior that it was possible to craft a single packet or packet stream which would be interpreted in completely different ways by the different hosts [52, 30]. Thus, any system behaving as a firewall or intrusion-detection system must simulate the behavior of *all* systems under its protection, lest the protector inadvertently allow packets through that tickle a vulnerability in one [55].

SQL injection vulnerabilities [44] are yet another example of the importance of parsers: code within the application combines user input with preconstructed SQL query fragments under the assumption that the input contains no single-quotes. It is too easy to skip parsing here—after all, the input is ASCII text and the query is ASCII text, so what parsing is necessary?—but the input is implicitly blessed nonetheless and therefore trusted by the query engine. Actively parsing the user input fixes this class of vulnerability, not least because it forces the developer to explicitly encode the assumptions the rest of their code is making [48].

All systems that accept input—that is, *all even-vaguely useful systems*—implicitly or explicitly include parsers. These parsers verify the correctness of the input before passing it to other components that process said input. If the parser is not correct, it will permit incorrect input to pass through, and the other components will perform incorrect operations. At best, these incorrect operations waste time; at worst, they allow a clever input-provider to take control of the entire system. So how can we make parsers better?

1.2 A Better Way

To guide our steps towards producing more-secure parsers, let us consider what is involved in the code that implements parsing.

First, we need to define the data structures that reflect the structure of the protocol messages. An instance of such a data structure is, in fact, the result of parsing, as shown in Figure 1.1. Furthermore, the consumer of the parsed data (be it the kernel, in the case of, e.g., IP or USB; or a userland program in the case of an application-level protocol) needs some way to refer to the various parsed fields. Since the overwhelming majority of kernel code remains C, collecting all those fields together in an instance of a C struct is the natural solution. Additionally, a C struct is the easiest basis from which to create a C++, Java, or Python object should the data consumer prefer a different format.

```
pass in quick on em0 proto tcp from any to any port 22 to 10.0.1.6
pass out quick on rl0 proto tcp from any to any port 80
pass in log on em0 proto {tcp,udp} from any to any
```

Figure 1.2: Policies written for the `pf` IP firewall. Note that they include both a description of the data using protocol-specific terminology (i.e., protocol type, port number, IP address) as well as an action (e.g., "pass in") to take if the description matches.

With types and corresponding data structure defined, we then need code that performs the actual parsing: it must read in a bytestream or a fixed-size frame and produce an instance of the protocol-message data structure. It should verify that the fields of the data structure conform to the requirements laid out in the protocol specification. If the input violates the rules of the protocol, the parser should report an error. Otherwise, the parser should return a well-formed member of the given type.

In addition to ensuring conformance to the protocol specification, ideally any parser would also support the ability to apply a user-specified filtering policy to incoming data, to reject certain inputs known to be troublesome. The canonical instance of such an ability is an IP firewall such as NetFilter [3] or `pf` [10], which allow a system administrator to describe the kinds of IP packets that should and should not be allowed passage. This may be a surprising entrant on the "need" list, but it shouldn't be: if decades of observing the security of communication protocols has taught us anything, it is to expect the unexpected. Baking-in the ability to respond to vulnerability disclosures by deploying a new policy rather than deploying an entirely new patched kernel makes defending vulnerable systems infinitely more practical.

Given such a filtering policy, then, we need to enforce it. At some point in the parsing process, the parser needs to compare the input data against the user policy and, if the policy matches, take the prescribed action. (Filtering policies frequently are of the form "take such-and-such an action if the input matches such-and-such a pattern". Examples of policies written for the `pf` firewall are shown in Figure 1.2.) The language used to specify user-defined policies should closely mirror the underlying protocol so as to make both writing and understanding the policy easier for mere humans.

With these components in hand, one might think our task is done. It is not. In addition to being able to parse data and enforce policy, we must be able to *test* the system. Without the ability to test it, we can have little or no confidence in its correctness under benign circumstances, let alone its behavior when targeted by evildoers. We need two specific tools to fulfill this need.

First, we need the ability to craft and inject arbitrary traffic into the parser. This task is

somewhat complicated by the fact that these parsers are often going to be running inside the kernel. Given that many of the data sources whose parsers need to be tested will be hardware-based (e.g., network interface controllers, USB devices, Thunderbolt devices), hardware might be required for injection. Buying a bunch of USB devices from your local electronics store isn't sufficient because (ideally) those devices will attempt to conform to the protocol specification. We also need to create and inject data that *doesn't* conform to the specification and verify that the parser correctly rejects it.

Second, we need to be able to monitor the behavior of the parser while it is performing its task. That is, the ability to debug the parser should be built into itself. Once again our task is complicated by the fact that this code will often be running inside the kernel: we must balance the need for efficiency with the need for transparency. Fortunately, there exist systems that balance these requirements quite well, which I describe later.

Thus, to summarize, we need:

- type definitions that represent protocol primitives (i.e., messages)—practically speaking, for kernel code, these take the form of C structs;
- code to parse raw binary data into the aforementioned data structures (which must also recognize when raw binary data fails to conform to rules regarding well-formed messages within the protocol, and reject it);
- a way for users to specify protocol-specific policy that is orthogonal to the syntactic correctness of the protocol itself (i.e., firewall rules);
- a way to enforce this user-defined filtering policy (which requires knowing where in the aforementioned code to place the hooks that enforce user-defined policy);
- a way to craft and inject arbitrary data, so as to test an implementation; and
- a way to observe the code as it handles the injected data, so as to locate and fix bugs—an instrumentation framework that exposes control flow within affected subsystems of the kernel.

With these tools in our pocket, we can proceed with relative confidence that our parser is, if not secure, at least rooted in principled development practices, flexible to unexpected needs, and suited to debugging in the face of misbehavior. This claim is intentionally weak: why should anyone believe the code I happen to write is any more secure than other code written by someone else? The answer is: they shouldn't.

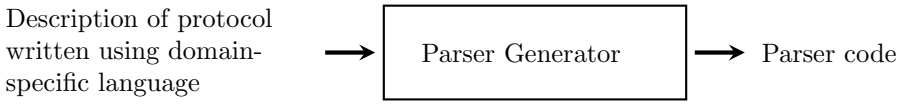

Figure 1.3: Workflow of a DSL-based parser generator. Given a description of the protocol language written in the domain-specific language, the parser generator produces the parser code.

Fortunately, besides the empirical evaluation against a suite of known vulnerability triggers, two separate methods exist to instill confidence in the security of such code: automagic generation of parsers (i.e., "parser generators") and formal verification. I will discuss each of these in turn.

1.3 Automagic Generation

As shown above, parsers and related tools are an important part of the trusted computing base. One strategy to improve security in general is to reduce the size of the trusted computing base such that it's feasible for a (small) set of people to manually audit. To that end, efforts have arisen that attempt to minimize the amount of code related to parsers, many resulting in domain-specific languages (DSLs) to describe protocols and *parser generators* to generate the actual parsing code. The idea is that the DSL protocol description and the parser generator—which is essentially a compiler from the DSL to, e.g., C—will individually be simple enough to audit. Figure 1.3 shows a stylized rendition of this workflow.

A domain-specific language is, as the name implies, a language designed to address a particular need. In contrast to general programming languages like C and Python, domain-specific languages are much narrower in scope, with both smaller specifications and smaller compilers. Additionally, domain-specific languages often need not be Turing-complete, which suggests that automating their verification may be more tractable [62]. For the purposes of this dissertation, a domain-specific language for protocol analysis provides a way to declaratively—as opposed to procedurally—describe the messages and state machine(s) for a given protocol as well as a mechanism to translate that description into executable code that parses messages of that protocol. In short, a language and a compiler for it.

1.3.1 Prior Work: DSL-based Parser Generators

This dissertation is not the first to use a domain-specific language to generate parsers. The classic example is `yacc` [34], which historically was used to generate parsers for programming language

source code rather than messages in a wire protocol. Given a grammar describing the programming language, `yacc` produces an LALR parser for that language, which can be augmented with code to execute when particular rules are matched.

DSLs have also been used in the realm of more traditional communication protocols. In 1998, Vern Paxson introduced the Bro [49] intrusion detection system (IDS), one of whose primary features is a domain-specific language for describing TCP/IP traffic. Its intent was (and continues to be) to run on a standalone machine on a network, examining all traffic for suspicious patterns. As such, it is not integrated with the running kernel and it is optimized for applying a variety of acceptable patterns to a given message rather than verifying the correctness of a single message in the context of a single machine. (Bro was, incidentally, inspired by an observation similar to the one attributed to Ptacek and Newsham in Section 1.1.).

More recently, GAPA [6] advertises itself as a "second-generation generic application-level protocol analyzer". While its intention is slightly orthogonal to the task of parsing protocol traffic for kernel consumption—it's targeted at rapidly creating vulnerability signatures to incorporate into userland intrusion-detection systems—GAPA has some lessons to teach. First, a quick, intuitive description language is important to encourage adoption. Second, while they originally intended for vulnerability detection, they realized that the ability to clearly and concisely describe protocol messages was useful in other areas, as well. They specifically mention tcpdump [33] and Wireshark [15] (née Ethereal), but the lesson applies even further than they claim: why not use this power to improve the kernel itself?

PADS [24] comes from the programming-languages research community rather than the systems or security communities, but has much the same idea as the other projects described here, with one significant difference: it is intended for parsing data that does not necessarily follow the strict rules of a protocol specification. The authors found themselves needing to parse data that frequently deviated from a normal, expected, easily-describable pattern, and therefore designed a DSL that produced parsers that were resilient to these deviations. This desideratum lies in stark contrast to the other systems described here, which exist precisely to ensure protocol traffic conforms to a standard. I explore the significance of this shortly.

To round out our mini-survey, Packet Types [39] is perhaps the closest in spirit to my work. The authors recognize that the task of parsing can be equated to the idea of testing whether a given message is a valid member of a particular type, where a sufficiently descriptive specification of the type can ensure the correctness of the message. Additionally, the authors draw explicit inspiration from functional programming languages such as ML, which is similar to my work's inspiration from

Haskell. Packet Types, however, does not attempt to integrate with existing operating systems, though they claim it would likely be possible.

Additional research that bears mention includes Shield [63] and binpac [47], both of which use domain-specific languages to generate parsers for wire protocols. Their contributions are superseded by the aforementioned tools, however.

The domain-specific languages and associated systems surveyed here, while not an exhaustive list, are representative of the state of the art within the systems and security communities. They do, however, have two significant shortcomings. First, all of the systems are intended to be run as userland applications, independent of running kernels. In the case of an intrusion detection system, this is not necessarily a fatal flaw (though defense in depth—i.e., validating protocol messages both at a network-wide IDS and at each individual host—is a Good Thing). When considering protocols like USB, however, in which devices are plugged directly into hosts, one cannot feasibly offload protocol analysis to a separate machine.

The other drawback of these systems is that, while they all feature extensive testing, none of them offer guarantees or proofs of correctness. Given the importance of parsers as described earlier, this is an unfortunate omission. Ideally, parsers would be subject to formal verification (described below, in Section 1.4). Before delving into that world, however, let us consider another way to specify a domain-specific language, one that does not require inventing a brand new language.

1.3.2 Embedded DSLs

The aforementioned systems created DSLs and their associated compilers from scratch. An alternative is to use an existing, possibly general-purpose, language and its compiler to achieve the same goal, enhancing and extending them as necessary. This approach is called an *embedded domain-specific language* (eDSL), because the domain-specific features are grafted onto (i.e., embedded into) another language and its compiler. Due to their already-declaratory form, functional programming languages are especially popular bases for eDSLs. Additionally, given the observation above regarding the job of the parser being to annotate incoming data with eloquent type information, a functional programming language that provides a rich type system is ideal. Both of these desiderata scream for a Haskell-like language.

There exist many examples of embedded domain-specific languages, and many specifically using Haskell. For instance, Parsec [37] is a library that provides a DSL for defining parser-combinators. It differs from the work in this thesis in that it attempts to be a general-purpose parsing engine, and

it does not produce code that can be natively integrated into a production kernel. Diagrams [23] is a Haskell-based eDSL for creating vector graphics.

Wang used Haskell as the basis of her Protege system [64, 66], which is similar to the work in this thesis in that it embeds a domain-specific language for describing networking protocols and generates parser code from. It differs, however, in that its primary target is embedded (i.e., hardware) systems, it does not attempt to support user-written policies or injection, and it does not make any claims about correctness beyond those derived from being an eDSL-based system. Furthermore, the protocol used in Protege's proof of concept—Modbus—is an entirely different domain to the protocol I used for my proof of concept: USB. That said, my work does not handle layering and encapsulation, and can take inspiration from Protege's implementation of same.

The hypothesis, then, is that we could embed a parser-generator DSL in Haskell to produce the various code artifacts described in Section 1.2. Therefore, to audit our system, we need only review the DSL description of the protocol in question and the Haskell code that "compiles" the DSL to C. We will have thus reduced the size of the trusted computing base, made the job of auditing the code involved in our parser much more tractable, and hence emerge with much more confidence in the security of our parser.

This is a significant win, but we can go one step further.

1.4 Formal Verification & Complete Mediation

For many years, the systems and security research communities have been using domain-specific languages (DSLs) to produce protocol parsers for use in operating system kernels, firewalls, and intrusion detection systems [6, 65, 24, 39]. These systems reduce the trusted computing base (TCB) of protocol parsers in the sense that, if one trusts the parser generator to behave as it claims, one need only ensure the correctness of the more-concise DSL description of the protocol. *If one trusts the parser generator to behave as it claims.* Casting no aspersions whatsoever on its authors, one would be remiss in tacitly trusting in the correctness of *any* software without formal proof to that effect.

Such proofs are the purview of the software verification community, which generally has a reputation for avoiding low-level systems code. The NICTA group in Australia recently demonstrated, however, the feasibility of producing a fully functional, fully verified, fully performant operating system kernel, l4.verified [35]. This project used a combination of hand-written Haskell, hand-written C, and hand-written proofs for the Isabelle proof-checking environment to verify security properties

of the resultant kernel. Other researchers in the verification community have also turned their attention to low-level code (as evidenced by, e.g., RockSalt [42] and Idris [7]), but it is far from the norm.

Therefore, on the one hand we have domain-specific languages feeding parser generators to ease the creation of parsers and on the other we have mechanisms to prove the correctness of low-level systems code. It is the concerted opinion of this author that these two worlds should merge. To that end, this dissertation presents a domain-specific language for describing protocols, one whose compiler produces not only the code to parse the described protocol, but also the model to facilitate verifying the correctness of the code produced.

Beyond verifying that the parser correctly parses protocol messages, we must also make sure that every message received by the system is evaluated by the parser. This is the notion of *complete mediation* and is distinct from formal verification in that the latter can only make claims about how it processes the messages it actually sees. Both static and dynamic analysis can be used to ensure a system exhibits complete mediation; I survey some of those now.

Static analysis is a technique in which inert code (source code or compiled machine code) is examined for runtime properties—in this case, complete mediation. Zhang et al [67] used CQUAL to determine whether the hooks provided by the Linux Security Modules framework really guard every avenue by which certain vital kernel data structures are accessed. CMV [56] uses static analysis to verify that Java bytecode programs exhibit complete mediation relative to system-level resources provided by the Java Virtual Machine.

In the realm of dynamic analysis, which analyzes running programs, Klee [12] uses symbolic execution to explore possible execution paths of a program. Given that it tests exhaustively by design, Klee is guaranteed to test for complete mediation, though its scaling properties leave much to be desired. Though admittedly less comprehensive, an empirical method to determine whether a system exhibits complete mediation is to instrument the system with hooks at the important places, send a great deal of crafted data at the system, and verify that the hooks intercept all the data. Any demonstrable example of an input bypassing the hooks would disprove complete mediation; of course, lack of such examples is heartening though not conclusive. This is the purview of fuzz-testing, which has been widely adopted by the software industry as a means of vulnerability testing. As I said, this is less comprehensive, but it can be easier to execute and its merits are apparent in the broad adoption of fuzzing techniques for exploring system security.

1.5 Practical Matters

The preceding sections outline a number of desired security properties for parsers (and, more generally, protocol stacks): specific technical artifacts whose code is both automagically generated and amenable to formal verification. The primary contribution of this dissertation is conclusive evidence that these are feasible in the context of a production-level operating system. In the following chapters, I describe the parser construction methodology and the various technical artifacts united by this methodology that I have produced to this end. But first, a brief description of the protocol I chose for my case study (USB) and the operating system with which I chose to integrate (FreeBSD).

1.5.1 Universal Serial Bus

Most security research involving parsing focusses on traditional networking interfaces such as Ethernet and TCP/IP. The Universal Serial Bus (USB), which most users take for granted as being used for keyboards, mice, and thumbdrives, was similarly taken more or less for granted by the security community—until about 2010 when our and other researchers' results demonstrated that vulnerabilities in USB implementations could be easily and effectively developed [8, 22, 17, 38]. The code that implements the USB protocol runs with the same kernel privileges as the code that implements TCP/IP, although exploitation in the case of USB requires physical access.

Not just that, but the very "universal" nature of USB makes it inherently more difficult to secure: the protocol is explicitly intended to support arbitrary devices. Like the TCP/IP stack, then, the USB protocol is layered such that arbitrary application-level protocols can be transmitted over the USB. The scary difference between the two, however, is that the host-side code that handles the application-level protocol *runs within the kernel*.

Therefore, a new kernel device driver is required to support a new USB device. Not only do device drivers run within the kernel (and hence with full kernel privileges) but they happen to exhibit surprisingly high vulnerability rates [14]. The large variety of USB devices induces the creation of a large variety of associated device drivers. It is highly *un*likely that these drivers have all received the same degree of exercise (and, by extension, auditing) as, for example, the USB keyboard driver. It is thus imprudent to believe that the collection of USB device drivers that ship with a given kernel is free of vulnerabilities. This is unfortunate, but not necessarily fatal: how many instances of obscure devices actually exist that can take advantage of these potentially-vulnerable drivers? It turns out not to matter.

The USB protocol begins with a process called *enumeration*, in which the host queries the device

for its identity, so that the host can associate the correct application-level device driver with it. With a customized device, an attacker can tweak the enumeration process to identify itself as any arbitrary device and thus pick precisely which device driver within the kernel to communicate with. An attacker would presumably target a driver with a known vulnerability; a defender would want to explore the behavior of drivers with potential vulnerabilities. The data structures exchanged during enumeration are complex and must be parsed; this is a known source of vulnerabilities [20].

To the best of my knowledge, very little effort has been expended to secure USB. In discussing this seeming deficit with employees of Microsoft who work in this area, the prevailing notion is that USB is limited to local attacks and are thus worth less attention than remote attacks to which, e.g., the TCP/IP stack is exposed. While this is true, it undersells the insidiousness of USB as an attack surface. An attacker can walk up to a powered-off machine, turn it on, stick in a USB device, and immediately have a direct line to the kernel in the form of USB device enumeration.

Two anecdotes provide compelling evidence to support the claim that USB is worth attention.

First: Stuxnet, the malware used to compromise the Iranian nuclear facility at Natanz. Following sound network defense principles, the uranium enrichment centrifuges were on a physically separate network from the outside world. That is, an airgap prevented attacks originating on the Internet from affecting the machines performing the uranium enrichment. Despite this, those protected machines were compromised; experts are convinced that USB was the vector by which those machines were infected [36]. Someone plugged a USB device into a machine that was otherwise protected by the airgap and the infection spread from there.

Second: in March 2013, Microsoft issued a patch for Windows that "could allow elevation of privilege if an attacker gains access to a system" [19]. CERT is more forthcoming in their description: "The USB kernel-mode drivers in [many versions of Windows] do not properly handle objects in memory, which allows physically proximate attackers to execute arbitrary code by connecting a crafted USB device" [13]. The disclosure goes on to discuss that the vulnerability exists in the code that handles device enumeration.

These anecdotes are not intended as criticisms of any party involved. Rather, they are evidence that USB is generally underappreciated as an attack vector and hence merits attention. The tools to deliver such attacks are not figments of our imagination: various USB hacking tools use the Teensy [61] development board to deliver scripted exploitative payloads via USB. A versatile open-hardware platform, USB Armory [4] far supersedes these capabilities, adding a full-featured microprocessor behind a USB interface.

More generally, USB is representative of line-oriented protocols (e.g., transport protocols such

as Thunderbolt and Fibre Channel; disk protocols such as SCSI) and thus the system I have implemented provides a model for how those protocols might be implemented more securely, as well.

1.5.2 FreeBSD

I chose FreeBSD [51] as the kernel whose USB stack to augment with the aforementioned tools. FreeBSD is a widely-deployed, high-performance kernel in the UNIX tradition, with a well-deserved reputation for clean, understandable code and carefully-architected subsystems. Integrating with FreeBSD demonstrates the viability of my approach in the context of a production-quality kernel (i.e., not just an academic toy).

1.6 Summary of Contributions

Thus far, I described the irreplaceable role protocols play in modern computing systems and the role parsers play in their implementation. I introduced the idea of using domain-specific languages to generate parsers, with the goal of reducing the trusted computing base to the DSL specification of the protocol and the DSL compiler. I contrasted the DSL approach with that of formal verification and argued that the two can (and should) be considered complementary strategies to producing trustworthy parsers.

In light of this, the contributions of this dissertation are as follows.

I developed a methodology for designing and implementing secure parsers and successfully applied it to the USB protocol in a production kernel. I empirically demonstrated that the case-study implementation is stable, effective in mediating malicious and non-standard inputs, and applied an industry-standard test suite to it. Due to their method of construction—autogeneration from type definitions of protocol messages and their elements—my USB parser/firewall is amenable to the same kind of automated verification that produced the fully formally-verified seL4 microkernel. I do, however, relegate such verification to future work as that effort is beyond the scope of a one-person project.

These contributions are presented in this dissertation as follows. First, I present (above) a detailed checklist of the components that ought to be implemented when supporting a new protocol (Section 1.2). Second, I present the tools I wrote to fulfill the needs of USB injection (Chapter 3) and inspection of a running USB stack (Chapter 4). Third, I present a case study I performed wherein I automagically generated a front-end parser for the USB protocol and integrated it with FreeBSD's production USB stack (Chapter 5). Finally, I empirically evaluate the security and performance of

my implementation using an industry-standard USB test suite (Chapter 6), demonstrating that it succeeded on all tests. In each chapter, I suggest by whom and where in the process of implementing a protocol the steps described therein should be performed.

In short, I demonstrate that it is feasible to construct parser generators whose products parse real-world protocols, are amenable to machine verification, integrate cleanly with existing operating systems, and are sufficiently performant to make a compelling case for their use. My study demonstrates that a similar approach can and therefore should be used in all security-critical systems.

Chapter 2

Overview of USB: Protocol and Vulnerabilities

This chapter provides an introduction to the USB protocol from the perspective of someone wishing to explore the attack surface presented by a host with working USB ports. Following the description of the protocol itself, I review the most comprehensive study to date of USB vulnerabilities, produced by the NCC Group in 2013 [20], which includes a classification of vulnerability types. Then I examine all USB-related vulnerabilities reported in the National Vulnerability Database [59] and attempt to place these vulnerabilities in the classification scheme proposed by the NCC work. The ultimate intention is, in later chapters, to show how the USB parser/firewall I produced can guard against these classes.

2.1 The Protocol

When a USB device is plugged into a *host*, the operating system running on that host communicates with it by sending requests and receiving responses. In the majority of circumstances, all communication is initiated by the host in the form of polling. (Low-level timing considerations are handled directly in the USB controller hardware and are below the level of abstraction I focus on in this work.)

When a device is initially plugged in, the host must query it to determine its nature—whether it is, e.g., a keyboard, a mouse, a MIDI device, or a printer. This initial conversation between host and device is referred to as *enumeration* and happens for every device. In addition to determining

```
Host:   80 06 00 01 00 00 12 00
Device: 12 01 00 02 00 00 00 40 1E 04 02 04 00 01 01 02 03 01
```

Figure 2.1: A request from the host for the first 18 bytes of the device descriptor (id 1, byte 4) and the device's response.

characteristics of the device such as polling frequency, preferred data transfer size, and power requirements, the host will also decide which kernel device driver to associate with the device when enumeration is complete. The communication channel that carries the enumeration messages is separate from application-level communication channels and is retained throughout the connected lifetime of the device. Once enumeration has finished, however, the associated device driver controls its own communication channels (called *endpoints*) to and from the device.

Note that this gives the device significant leverage over the operating system: it gets to pick and choose precisely which driver to handle its application-level data. In essence, data sent by the device determines the code paths and control flows that handle data sent henceforth. If an old, poorly maintained, buggy (i.e., vulnerable) driver is still shipped with an operating system, one could use a custom USB hardware device to select it during the enumeration process and exploit it.

Note also that plugging a USB device into the machine gives an immediate communication channel direct to the kernel. Even following enumeration, most application-level USB drivers still run in kernel mode (though this is changing: see Microsoft's User Mode Driver Framework [18]) as well. Despite the direct line to the soft, defenseless innards of the operating system, I know of no framework—prior to mine presented here—that defends against attacks by this vector.

Returning to enumeration, most of the messages that comprise this process are *descriptors* that contain various parameters of the device in question. Figure 2.1 shows a request for a descriptor sent from the host to a device and the response containing the descriptor itself. In this example, the fourth byte of the host's requests identifies the descriptor being requested (in this case, the "device" descriptor) and the seventh byte indicates the amount of data the host would like to receive back (in this case, 0x12 = 18 bytes).

Then, in the response, the first byte indicates the total number of bytes sent by the device. This presents a classic opportunity for an exploitable bug. If the host does not verify that the received data is in fact 18 bytes long (in this case), then the host runs the risk of either underflowing or overflowing a kernel buffer. (The infamous Heartbleed [46] vulnerability is an example of an underflowed buffer and overflowed buffer examples are legion [45, 53].)

My work in this thesis focusses entirely on the enumeration phase of the USB protocol. Some might consider that limiting but, as we will see, this phase can harbor a surprisingly large number of bugs.

2.2 USB As a Gateway to the Kernel

The previous section covers enumeration but not application-level protocols. Given that the nature of USB enables a device to pick and choose precisely which driver within the kernel to exchange application-level data with, this is a notable omission. Allow me to address that.

Like the TCP/IP family of networking protocols, USB is *layered*: it allows data from one protocol to be encapsulated inside another. This is, in fact, precisely how USB supports such a wide variety of devices: once enumeration is complete, the active part of the USB protocol steps aside and mostly just ensures the delivery of application-level data between a collection of host and device endpoints through codepaths designated during enumeration. Since many USB devices implement application-level protocols that have been natively implemented in past (e.g., SCSI, audio, keyboards) this often provides a direct codepath to parts of the kernel outside the USB stack itself.

The work we presented at the Workshop on Embedded System Security in 2012 [8] explored the reachability of kernel logic from the USB interface with a focus on the storage subsystem, down to the granularity of basic blocks. We found that a USB device could access essentially the entire FreeBSD storage subsystem, which is particularly notable because so many other aspects of the system depend on disks. Furthermore, this is only the storage subsystem. We conjectured that our results could extrapolate to the many subsystems in the kernel proper likely touched by USB devices, including printing, networking, and human-interface devices.

Therefore, the fact that so many codepaths in the kernel are accessible via USB only increases the importance of correctly parsing the data that arrives from untrusted devices. It is a crucial boundary to ensuring the security of running systems.

2.3 Vulnerabilities

In 2013, Andy Davis of the NCC Group wrote a test suite, umap, to comprehensively explore the behavior of operating systems in the face of unexpected input received during USB enumeration. He published his results in an aptly-named technical report, *Lessons learned from 50 bugs: Common USB driver vulnerabilities* [20], which I summarize here. It should be noted that umap builds on the

injection framework I constructed as preparatory work for this thesis, which I describe in Chapter 3. Thus, my initial work on this thesis enabled the creation of an industry-standard security testing suite.

Using special-purpose hardware driven by Python scripts, he emulated a variety of USB devices being plugged into a target host and controlled every aspect of the data sent from these devices to the host during each enumeration phase. His scripts caused the emulated devices to send intentionally malformed data to the host while he observed how the host responded—a crash indicated that the host does not correctly handle the malformed data. He tested a large variety of malformations and produced the following ontology of bugs.

Unspecified Denial of Service, in which the driver or host machine usually crashes, but not in a way that is exploitable by an attacker. This class includes null-pointer dereferences and out-of-bounds reads. Davis does not consider these security-related bugs in the context of USB drivers.

Buffer overflows, in which bounds are not adequately checked prior to memory operations. As an example, Davis described a Linux driver that allocates 80 bytes for the contents of a string descriptor; but the string descriptor can be up to 252 bytes long. Thus, when a string descriptor longer than 80 bytes arrives, the remaining data overwrites other kernel memory, which is certainly a bug and quite possibly exploitable.

Integer overflows and other length-related bugs, in which arithmetic performed on numbers provided by the device can lead to unintentional memory allocations. Consider the bug described in the previous paragraph: the logical solution would be to read in the length of the descriptor (an 8-bit value), allocate the appropriate amount of memory, and then copy the string into that memory. If, however, any arithmetic is performed on the length, an attacker could cause the length to overflow past 255 and cause *less* memory than necessary to be allocated. Then, when the string is copied, it could again overwrite existing data structures. Davis identifies instances of this happening in hub descriptors, configuration descriptors, endpoint descriptors, HID descriptors, image class data transfers, and printer class data transfers.

Format string bugs, in which user-controlled input is used as the format string in calls to the `printf` family of functions. Historically, this allowed an attacker to write to arbitrary locations in memory using the "%n" format specifier. While this specifier has been widely deprecated, many compilers still support it. (In fact, Kees Cook demonstrated just such an attack using a custom USB device in 2012 [17].)

Logic errors, in which the operating system incorrectly handles a given input. As Davis points

out, these are implementation-specific because the logic of driver code often varies greatly between operating systems. Some such logic error result in memory corruption when an 8-bit field is set to `0xFF`, where the protocol specification only expects values between 0 and 127. In this case, the operating system is incorrectly handling unexpected input.

2.3.1 National Vulnerability Database (CVEs)

A kind of "ground truth" of vulnerabilities in deployed software is captured in the National Vulnerability Database. Vulnerability disclosures, dubbed "CVEs" (Common Vulnerabilities and Exposures), are reported and assigned on the basis of particular products and technologies found to be vulnerable. The CVE system does not attempt to classify vulnerabilities. Recently, an attempt to provide an ontology of the underlying causes for CVEs has been made in the form of the Common Weakness Enumeration (CWE) system [58]. For our purposes, however, the NCC classification described above is more suitable, being targeted specifically to USB.

Between January 2005 and December 2015, exactly 100 of the vulnerabilities reported to the NVD contained the string "usb". I surveyed all 100 of these vulnerabilities and placed each in one of the five categories identified by Davis. I use the result of this survey [1] in Chapter 6 to gauge the effectiveness of the USB firewall I created based on the distribution of bugs between classes. Not surprisingly, the bugs mitigated by proper parsing form a significant subset.

2.4 Realization

While a world in which every protocol designer, implementer, and user is intimately aware of both the details of the protocol and existing bugs in various implementations would be fantastic in the sense of being wonderful, it is also fantastic in the sense of being unrealistic. At the very least, however, the designer and implementer (both client- and host-side) should have such knowledge. Additionally, implementation maintainers should remain abreast of security disclosures relative to the implementations they administer.

[1] The raw results are in Appendix A.

Chapter 3

Injection

> "Security won't get better until tools for practical exploration of the attack surface are made available." (Joshua Wright, 2011) [28]

The first piece of my research plan was a framework for injecting arbitrary, crafted frames into USB.

Despite the security benefits promised by formal verification, the ability to empirically evaluate the security of deployed system remains vital; this ability rests on being able to craft and inject arbitrary traffic. For starters, no fully-verified systems have been deployed in any meaningful quantity. Additionally, an enormous number of *un*verified systems exist in the wild whose security we need to be able to analyze.

Additionally, the benefits espoused by the formal verification community are, perhaps, not as widely-applicable as we might like to think. To see why, consider that formal verification attempts to prove that certain properties of code are maintained (e.g., that execution proceeds linearly from instruction to instruction except in the case of explicit, intended branches). These statement of formal correctness are only useful if we are able to completely enumerate *all* relevant security properties. For the most generic software models, the task may, in fact, be impossible [54].

As a side-note, symbolic execution is a complementary approach to formal verification that attempts to explore how a system behaves in the face of all possible inputs. The state of the art in this area, Klee [12], is effective for a wide variety of software, but it suffers from scalability issues due to the massive state-space explosion incurred by branch-heavy code [11].

Due to these concerns, many security practitioners feel that we cannot trust the security of large-scale systems without thorough probing of their attack surface. This notion is embodied by the quote

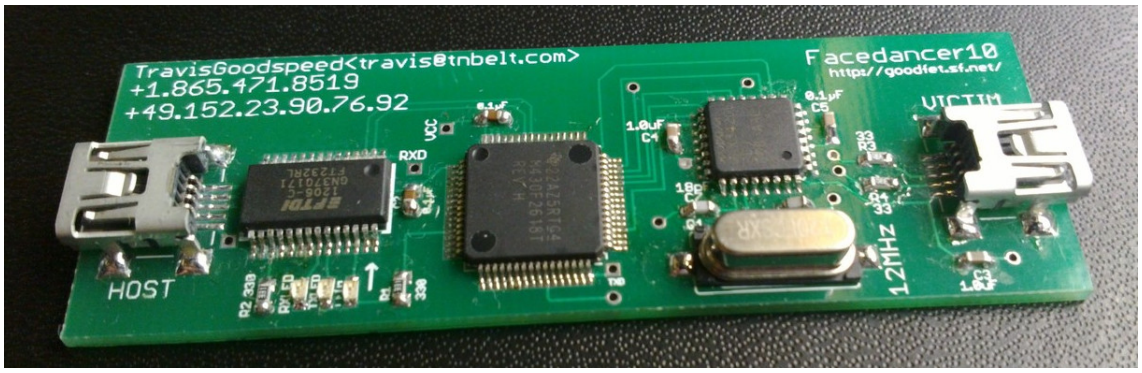

Figure 3.1: The Facedancer board (version 10).

above, sometimes referred to as Wright's Principle. This dissertation accepts this assertion: namely, that the ability to inject arbitrary data into that system is crucial. While commercial solutions to inject arbitrary bits onto a USB exist (e.g., MQP Packet-Master USB-500 [43]), it was not until Travis Goodspeed produced the Facedancer [27] that this capability was accessible to a widespread audience.

3.1 Facedancer

The Facedancer platform—the board designed by Travis Goodspeed in collaboration with Sergey Bratus, with software contributed by me—is a custom PCB based on the GoodFET, a general-purpose, open-source JTAG adapter. As seen in Figure 3.1, it connects to both a host and a target (victim). The host sees the connected Facedancer as a standard USB FDDI (serial) device which can be controlled using a simple datagram-based protocol, whereas the target sees whatever USB device the host chooses to emulate.

This emulation is driven by commands received by the Facedancer over the serial line. A dedicated FDDI chip passes commands received on to the Texas Instruments MSP430 microcontroller that serves as the main computational brains of the board and speaks the generic GoodFET serial protocol. Commands that request sending or receiving USB frames to or from the target are passed over an SPI bus to the MAX3421 USB controller chip [31], which is connected to the target-facing USB type-A port.

The MAX3421 handles much of the low-level, time-sensitive aspects of the USB protocol—for instance, responding with NAKs while the host-side code is composing a response to a request sent by the target machine over the USB. The presence of the MAX3421 chip thus allows the host to

```
USBDevice
+-> USBConfiguration
| +-> USBInterface
| | +-> USBEndpoint
| | +-> USBEndpoint
| +-> USBInterface
|   +-> USBEndpoint
|   +-> USBEndpoint
+-> USBConfiguration
  +-> USBInterface
    +-> USBEndpoint
    +-> USBEndpoint
    +-> USBEndpoint
```

Figure 3.2: The USB hierarchy as implemented in the host-side Facedancer stack. This figure describes a single USB device that supports two distinct configurations, the first consisting of two interfaces, each with two endpoints, and the second consisting of a single interface with three endpoints.

focus on the content of its responses rather than, e.g., timing issues on the bus.

3.2 Host-side Software

The Facedancer comes with a host-side Python library for emulating USB devices. As such projects are wont to do, however, it evolved to meet the needs of debugging in-development hardware until the hardware platform stabilized. The original code was thus poorly organized and poorly documented. Furthermore, it did not reflect the structure of the USB protocol: in short, it was difficult to adapt to arbitrary uses, which was the main purpose of the project to begin with. Therefore, I wrote a brand-new host-side software stack for the Facedancer from scratch.

3.2.1 USB Component Hierarchy

My code follows the USB hierarchy of concepts with the intention that it be easier to understand and modify. An example instance of this hierarchy is shown in Figure 3.2. This hierarchy captures the notion that a single, physical USB *device* may have multiple *configurations* (i.e., sets of interfaces it may present to a host), each of which may have multiple *interfaces*, each of which may have multiple *endpoints*. These terms follow the established USB vocabulary.

Perhaps confusingly, the *interface*—in USB terminology—is what encapsulates the functionality of, e.g., a mouse or a keyboard. The notion of a device, on the other hand, is the physically-connected

object, which may present itself to a host as multiple logical devices. (Think of a USB keyboard with a trackpad: it connects to the host via a single cable, but the host recognizes it as two logical devices—a keyboard and a mouse. In this example, the *thing* that plugs in is a *device* whereas the keyboard and mouse are *interfaces*.) The operating-system implication of this division is that separate interfaces on a single (USB) device may be associated with different (kernel) device drivers. This separation has been used by a number of production designs, such as SanDisk's U3 technology and related DRM schemes.

In USB, *endpoints* are channels of communication, somewhat analogous to ports in TCP and UDP. (For additional analogies between USB and TCP/IP concepts, see Table 3.1, duplicated from our WESS paper [8]; further details may be found there.) They allow multiplexing communication over the single USB cable. Their primary use is to segregate device-level control messages from application data; the former uses endpoint 0 whereas the latter may use one or more of the remaining 255 interface-specific endpoints.

3.2.2 Enumeration

When a USB device is first connected to a host, the host and device carry out a process called *enumeration*, in which the former interrogates the latter as to its capabilites. This conversation consists of *descriptors* sent from the device to the host which describe all aspects of its behavior: configurations, interfaces, endpoints, strings, etc.

My code cleanly delineates between the logic that implements enumeration and that which implements application-level control. As such, it is easy to customize specific aspects of the emulated device. Instead of the raw bit-banging typical in prototypes, the new stack cleanly delineates the various functionality in a class hierarchy that mirrors the hierarchy of USB itself. Therefore, to customize aspects of a particular endpoint, one need only focus on the USBEndpoint instance that implements the functionality in question.

This is not to say that the new stack doesn't support bit-banging! A combination of callback functions and member variables allows arbitrary code to handle any aspect of the emulated-device-to-host communication, so a potential fuzz-tester is welcome to override the default functionality in any way he or she sees fit. One benefit of the class structure mirroring the structure of the USB protocol, however, is that *finding* the code to override is much easier. Instead of having to dig through code that manually assembles hard-coded binary strings to find where the configuration descriptor is created, one need only override the `get_descriptor` method of the `USBConfiguration`

USB	**Ethernet**	**Assumption**	**Violation**	**Attack Use**
Transfer	one round-trip, maybe NAK'd	Intended device will reply to the transfer	non-compliant controller	hijack session, change state under nose of host
Transaction	one set of transfers, all but the last NAK'd	host controller complies with USB spec on transactions	hijack session on disconnect	use of trusted session context
Packet	packet fragment	implicit length of concatenated frames will matfh explicit length of transaction	non-compliant device	memory corruption, info leak
Controller	Ethernet card	n/a	n/a	n/a
Bus	D+/D- pair	electrically legal signals, but in realize those widely outside of spec are accepted	non-compliant controller	damage frames for session hijack, jamming

Table 3.1: Analogous features between Ethernet networks—which we have much experience securing—and USB infrastructure. (Reproduced from the paper that presented this work at the Workshop on Embedded Systems Security [8].)

class.

As a result, creating a new emulated device or modifying an existing one is quicker, simpler, and more intuitive. It was gratifying to find this code become the basis for the industry-standard test suite, umap.

3.3 Reference Emulations

Using the library described in the previous section, I implemented programs that emulate a USB keyboard, a USB mass storage device, and a USB FTDI (serial) device.

For each, I created a new class that derived from the `USBInterface` class, in which I defined the static parameters of the device (e.g., number and type of endpoints, manufacturer and product strings). When this class is instantiated, the library code takes care of marshalling and sending the various descriptors during USB enumeration: no customization of device initialization is necessary. It just works.

Once enumeration is complete, the emulated device will need to handle incoming requests from the target machine. These requests are handled entirely by a single function within the `USBInterface` class. Therefore, to customize the behavior of the emulated device, one need only customize this particular function. Thus, the author of an emulated USB device is insulated from the complication of the USB protocol itself and is left to focus on the device-level protocol. (To wit: the source file that implements the USB FTDI device, `USBFtdi.py`, is entirely composed of code that deals with serial requests. The only USB-specific parts are those that define parameters of the device to be sent during the enumeration phase.)

This design benefits a programmer seeking to implement a well-behaving device as well as one seeking to implement a *mis*behaving device, for the purpose of probing the security of a host's USB stack. In this case, the programmer need only derive a new instance of the class which contains the functionality he wishes to customize and override the appropriate function. For instance, if a programmer wanted to explore how a host handles malformed configuration descriptors, he or she would derive from `USBConfiguration` and override the `get_descriptor` function of his new class. Then, when the target machine asks the emulated device for its configuration descriptor, it is sent a descriptor produced by the customzed code rather than the default, well-formed descriptor.

3.4 Code

All of the code described in this chapter is available in the public GoodFET repository on github: `https://github.com/travisgoodspeed/goodfet`, under the "client" subdirectory.

The library code described in Section 3.2 is in the `USB*.py` files.

Code that implements the logic of emulated devices described in Section 3.3 is in `USBKeyboard.py`, `USBFtdi.py`, and `USBMassStorage.py`.

Executables to run these emulated devices are in the `facedancer-*.py` files.

3.5 Realization

As illustrated by the quote that opens this chapter, the ability to craft and inject arbitrary traffic is vital to producing implementations whose security we can trust. All implementations, therefore, should be tested with such tools. Fortunately, because injection may (and likely *should*) happen external to the protocol implementation itself, injection tools can be developed separately from, e.g., operating system kernels and client hardware.

The most appropriate point in the protocol development pipeline where injection tools could be implemented is in parallel with an operating-system agnostic reference implementation, which is usually produced by the group developing the protocol itself. Failing that, unfortunately, there is no single, obvious party, which is why we have seen the independent security community taking the task on itself, in the form of third-party injection tools such as the Facedancer.

Chapter 4

Inspection & Instrumentation

This chapter describes the instrumentation framework I designed, built, and applied to the USB subsystem of the FreeBSD kernel. This framework had two goals: both to understand the existing structure of the USB stack and to evaluate the effectiveness of my parser/firewall. The former goal was interesting in its own right, but more importantly, it provided the insight necessary to integrate my USB parser/firewall with FreeBSD's USB stack. The latter was necessary for demonstrating the efficacy of my implementation.

Injecting custom-crafted data, as described in the previous chapter, is vital for exploring the security of systems. It does not, however, directly allow for the improvement of those systems because the feedback is usually not sufficiently specific to inform debugging, vulnerability mitigation, or vulnerability development efforts. A systems engineer needs to know precisely which code handles the data and how so that when crafted data causes runtime errors, the offending code can be found and fixed.

Mitigation plays an important practical role in operational security: specifically, in addition to fixing bugs—which a user might wish to protect against before the vendor ships a fixing patch—one might wish to implement user-defined policies on data flowing through the kernel. (Note that this is separate from but complementary to checking that data flowing through the kernel conforms to the protocol specification. As an example, one might wish to implement a policy that allows only HID devices like mice and keyboards to be connected via USB.) Where in the flow of data through the kernel should such policies be enforced? More specifically, where in the source code should enforcement hooks be placed?

The DTRACE [57] system available on FreeBSD and Solaris pioneered the very sort of fine-

grained observation of live systems that would achieve this goal. (SystemTap on Linux was inspired by DTRACE and fulfills a similar need.)

Unfortunately, even though DTRACE provides more insight into running systems, and with more granularity, than has previously been available, it didn't provide all the information necessary to guide debugging and enforcement efforts. Specifically, its Function Boundary Testing probes were unreliable; not only that, but they only fired when a function was called or returned: they did not communicate the flow of control *within* a function. When considering kernel functions that may be hundreds of lines long, this is a significant gap in observation.

Therefore, I implemented a set of custom, static DTRACE probes for FreeBSD that trace execution within its USB stack at the basic-block level. Additionally, I implemented a set of scripts that visualize these traces, showing the interaction between components of FreeBSD's USB stack, at the level of both functions and source files. These tools have a number of benefits: they provide a heretofore-unavailable view of control flow within the kernel, they help readers of the kernel code understand how the various components fit together (which can be useful to both kernel newbies and grizzled veterans alike), and they guide the placement of policy-enforcement hooks such that they can be most effective.

This work was presented at the Workshop on Embedded Systems Security in 2012 [8] and is summarized below.

4.1 Instrumentation

DTRACE is an instrumentation framework for monitoring running FreeBSD and Solaris systems. As such, it provides a large number of built-in probes that report on everything from system call invocations to disk I/O patterns to system clock behavior. They do not, however, include probes that let us trace the execution of kernel code at the basic-block level, which is necessary for the debugging and hook-placement tasks described above.

Fortunately, DTRACE allows one to create custom sets of probes: thus, I created the `usb_bb` probeset and defined the following probes within it:

- `MY_FUNC_ENTER(filename, function_name)` at the beginning of every function.
- `MY_FUNC_RETURN(filename, function_name)` at every point from which a function might return.

```
#!/usr/sbin/dtrace -qs

usb_bb:::enter,
usb_bb:::return
{
    printf("%s/%s (%d) %s %s %s %d\n", curthread->td_proc->p_comm,
        curthread->td_name, curthread->td_tid, probemod, probefunc,
        probename, arg0);
}
```

Figure 4.1: Example script, written in `D`, that enables custom DTRACE probes `MY_BB_ENTER` and `MY_BB_RETURN`; prints a message when execution reaches any of those points.

- `MY_BB_START(filename, function_name, index)` at the beginning of every basic block, where `index` uniquely identifies each basic block within the function.
- `MY_BB_FINISH(filename, function_name, index)` at the end of every basic block, where `index` matches the identifier of the associated `MY_BB_START` probe point.
- `MY_MUX(filename, label)` at every point where a control-flow decision is made based on an input value—in practice, this ended up being mostly function-pointer calls.

I manually added these probes to a local copy of the FreeBSD kernel source code. Table 4.1 summarizes the extent of these modifications within its USB stack.

probe	qty	event
`MY_FUNC_ENTER(file, func)`	204	upon function entry
`MY_FUNC_RETURN(file, func)`	356	upon function exit
`MY_BB_START(file, func, n)`	1235	upon starting basic block n in a given function
`MY_BB_FINISH(file, func, n)`	1235	upon finishing basic block n
`MY_MUX(label)`	30	immediately prior to invocation of a callback

Table 4.1: Summary of static probes in our instrumentation framework.

Once these probes are placed in the kernel, they are quiescent until activated by a script written in a special-purpose language called `D`. For example, the program in Figure 4.1 activates the `MY_BB_ENTER` and `MY_BB_RETURN` probes and prints a message when execution reaches any of them. When this script is run from the shell (with root privileges), it prints out one line per probe encountered and exits when the user presses Control-C.

In the course of my instrumentation, I implemented many `D` scripts; descriptions of the notable scripts follow.

- `bb-count.d` counts the number of times each instrumented basic-block executes.
- `bb-trace.d` prints a properly-indented message when a basic block is entered or left.
- `mux-trace.d` prints a message when a mux-point is reached, indicating the value of the data that caused the choice of subsequent execution path.
- `cam-trace.d` traces the execution of storage subsystem operations within the USB stack. I used this to help understand the structure and behavior of interactions between the generic FreeBSD disk layer and the USB mass storage device driver. (CAM stands for Common Access Method, a FreeBSD abstraction for interfacing with storage devices.)

Of these, the most interesting is `bb-trace.d`. Running it while performing USB-related activities results in a guide to how those activites are processed within the kernel. Figure 4.2 shows an example output. This output begins with a call to `usb_callback_proc` in kernel thread 100036. In the zeroth basic block of that function, `usb_command_wrapper` is called, which sees execution of its zeroth, third, fourth, fifth, sixth, and eighth basic blocks. This pattern continues until `umass_t_bbb_command_callback` calls `usbd_xfer_softc` which immediately returns, followed by a call to `usbd_xfer_state` which also immediately returns. Upon returning, execution within `umass_t_bbb_command_callback` resumes in the second basic block.

This record can be related back to the actual instrumented source code, thus allowing the observer to follow along in (somewhat) real time. Note that following along at this level of granularity is not possible with the default set of DTRACE probes on FreeBSD (and even the function-level granularity provided by the FBT probes is unreliable).

While undoubtedly interesting, the raw output of, e.g., `bb-trace.d` is less immediately useful. I wrote a collection of Python, awk, and shell scripts to process this raw data into a more useful format. (Another benefit to post-processing is to reduce the number of cycles required to process probes firing in real-time: adding more intelligence to the `D` scripts causes probes to be dropped.) The most interesting of these scripts is `indent-bb-trace.py`, which takes as input a basic-block trace such as the one shown in Figure 4.2 and produces an indented trace as shown in Figure 4.3.

```
100036 usb_callback_proc enter 0
100036 usb_command_wrapper enter 0 3 4 5 6 8
100036 usbd_callback_wrapper enter 0 4 5 13
100036 umass_t_bbb_command_callback enter 0
100036 usbd_xfer_softc enter 0
100036 usbd_xfer_softc return
100036 usbd_xfer_state enter 0
100036 usbd_xfer_state return
100036 umass_t_bbb_command_callback in 2 3
100036 usbd_xfer_get_frame enter 0
100036 usbd_xfer_get_frame return
100036 usbd_copy_in enter 0 1
100036 usbd_get_page enter 0 1 2 3 6
100036 usbd_get_page return
100036 usbd_copy_in in 2 3
100036 usbd_copy_in return
```

Figure 4.2: Output from the `bb-trace.d` DTRACE script, showing the basic blocks executed in each function.

```
-> usb_callback_proc 0
  -> usb_command_wrapper 0 3 4 5 6 8
    -> usbd_callback_wrapper 0 4 5 13
      -> umass_t_bbb_command_callback 0
        -> usbd_xfer_softc 0
        -> usbd_xfer_state 0
         umass_t_bbb_command_callback 2 3
        -> usbd_xfer_get_frame 0
        -> usbd_copy_in 0 1
          -> usbd_get_page 0 1 2 3 6
           usbd_copy_in 2 3
        <- usbd_copy_in
```

Figure 4.3: Indented basic-block trace. (The underlying data is the same as in Figure 4.2.)

4.2 Experimental Results

My enhancement of the FreeBSD DTRACE dynamic probe system allows the user to formulate and test hypotheses regarding reachability of specific parts of code—down to the granularity of basic blocks—by specific USB inputs. To the best of my knowledge, this granularity has previously only been available in static analysis tools such as IDA Pro [29], which are unsuitable for analysis of running systems. The fine-grained instrumentation described in the previous section provides a great deal of insight into the code being executed inside the kernel. What can we do with this insight?

4.2.1 Measuring Code Coverage

While testing code thoroughly is a worthy goal, all too frequently code is shipped that has not undergone sufficiently rigorous examination. When deployed, many pieces of software are implicitly tested by users interacting with their systems performing everyday tasks. In the best-case scenario, upon discovering erroneous behavior, a user will file a detailed bug report allowing the code's author to fix the bug.

This informal testing can be effective, but it is far from complete. The instrumentation framework described above can be used to measure how incomplete it really is. This achieves a tangible security benefit: knowing the well-tested (even if informally) code paths allows one to concentrate code auditing efforts on the less-used (and therefore less informally-tested) code paths. By virtue of undergoing less exercise, these latter paths are more likely to harbor potentially-exploitable bugs.

I used the `bb-count.d` probe script described above to gather the number of times each basic block in FreeBSD's USB stack was executed. I then inserted a USB thumbdrive (which uses the `umass` driver within the USB stack), read a single block of data, wrote a single block of data, and ejected the drive. Finally, I terminated the probe script. These actions exercised the normal code path for USB device enumeration, USB mass storage device initialization, read and write interaction between the USB stack and the FreeBSD storage subsystem, USB mass storage device finalization, and USB device removal.

By process of elimination, then, we can deduce the *ab*normal code path. To that end, Figure 4.4 shows the number of basic blocks exercised (and not exercised) per file in FreeBSD's USB stack during the experiment described above. I did not see any variation in this data over multiple runs; but variation would certainly be cause for concern! It would indicate that, despite inducing what should be a completely deterministic sequence of code events, some aspect of the system is

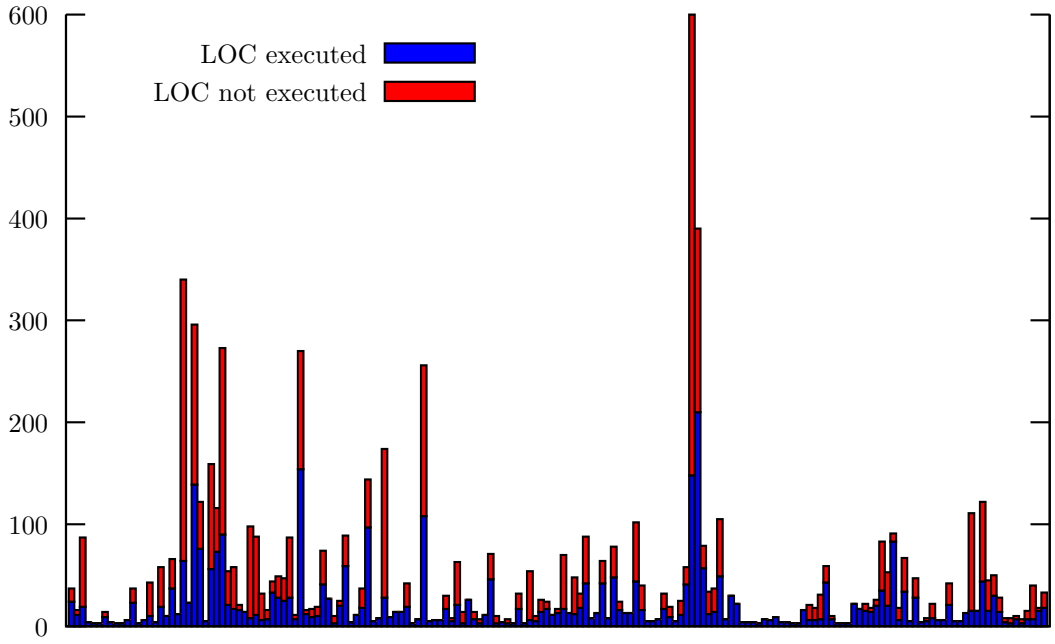

Figure 4.4: Lines of code exercised per-file during a USB thumbdrive insertion, read, write, and removal.

introducing irregularity: a potential malicious actor [26].

4.2.2 Guiding Hook Placement

A more specific use of the basic-block traces is to identify where in the data-processing control flow would be most advantageous to place enforcement hooks. That is, the function traces produced by the framework described in this chapter are caused by activity happening on the USB. If we want to make sure that said activity follows the rules of the protocol and/or conforms with a user-defined policy, we need to examine it. Code that implements this examination must be called at some point in the control flow revealed by these traces. But where?

This is a Goldilocks game of finding a place in the code that all USB events traverse. Ideally, for efficiency reasons, we want the enforcement hooks invoked no more than once per event. On the other hand, we *need* the enforcement hooks invoked at least once per event, otherwise enforcement will miss events and lose its power. The basic-block traces described in this chapter provide precisely the data necessary to discern where to put the enforcement hooks. These are described in detail in Section 5.5 along with the rest of the work involved in integrating the enforcement system into FreeBSD.

4.2.3 Showing Inter-component Interactions

Finally, I used the traces produced by my instrumentation of FreeBSD's USB stack to generate graphs of the control flow between components within the stack. The source code is divided into separate files by function (e.g., the controller interface is in `controller/ehci.c`, request handling is in `usb_request.c`, DMA operations are in `usb_busdma.c`); therefore, I chose source files as a reasonable approximation of "component". The resulting graph is shown in Figure 4.5.

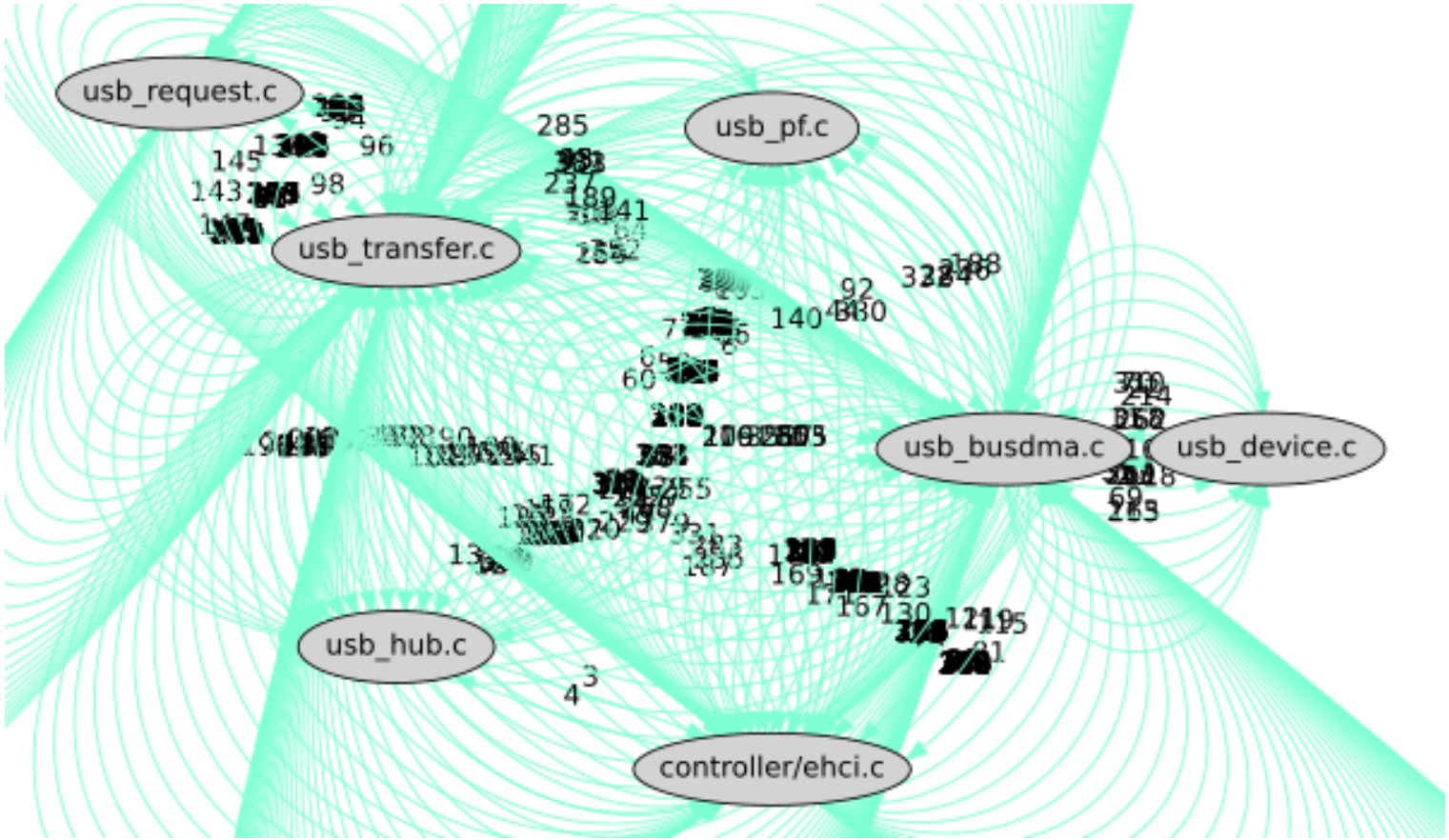

Figure 4.5: Diagram generated from basic-block trace showing interactions between different modules within FreeBSD's USB subsystem. Nodes represent source files within the FreeBSD USB stack; each directed edge represents a call from a function in one file (origin node) to a function in another file (destination node).

4.2.4 Summary

In this chapter, I describe how I instrumented the FreeBSD USB stack using DTRACE probes. I used this instrumentation to map the basic blocks within the stack that are exercised during normal operating conditions; by extension, I identified the basic blocks that *don't* get exercised, which therefore might be good places to focus a security audit. These efforts come together in my empirical evaluation of my parser/firewall's efficacy at stopping malicious inputs.

I also foreshadowed how I would use the results of the instrumentation to inform the placement of enforcement hooks that validate USB frames as they flow between host and device. Lastly, I present diagrams that describe the various inter-component interactions within the USB stack.

4.2.5 Realization

The instrumentation I describe in this chapter is part and parcel with the in-kernel implementation of the protocol. As such, the appropriate party to implement it is the kernel maintainers. We have seen this in the DTRACE system in FreeBSD; Solaris and its derivatives are similarly equipped. Linux has a similar framework called SystemTap. Microsoft Windows has a variety of monitoring systems built in [32]. All that remains, therefore, is for an enterprising kernel maintainer to write the code, which is itself fairly straightforward (though potentially voluminous).

Chapter 5

Generation

In Section 1.2, I offered a programme of code artifacts necessary to implement support for a protocol and claimed that, given a specification of the protocol in question, this code could be automagically generated. I intentionally eschew the term "automatically" because it does not encompass the structural qualities of the resulting code; rather, I use the word "automa*g*ically" to mean that the code exhibits additional properties that make it amenable to further automatic processing, such as formal verification. The distinction is not a trivial one; for example, without substantial effort on behalf of the programmer, the outputs of yacc and bison are likely not formally verifiable.

In preceding chapters, I describe my efforts to manually implement components that inject crafted USB data onto the bus and provide a view of the code touched by the injected data. This chapter describes the centerpiece of the programme, the autogeneration framework; the previous pieces are merely needed to support and test it. I describe what a protocol specification entails, how the code is generated, and how it is integrated into a production operating system kernel.

I claim that a great deal of code—in fact, much of the code that handles a protocol within the kernel—can be generated automagically from a specification of the protocol. Consequently, the ad-hoc code that performs these operations can be replaced in actual operating-system kernels and protocol stacks with generated code without appreciable loss of efficiency and with significant gains in security (e.g., ability to mitigate malicious inputs). I present a case study that demonstrates how this works for USB.

Specifically, I claim the data structure(s) that hold protocol messages, the code that parses messages from the wire into these data structures and verifies their contents, functions that access fields within the data structure, and functions that print the contents of a message in human-readable

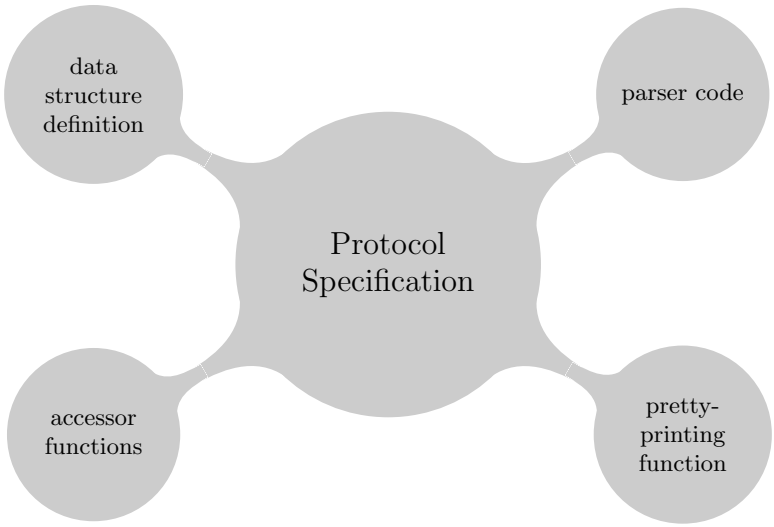

Figure 5.1: Given a protocol specification, shown in the middle, we can automagically generate the code required to implement support for that protocol, shown in the leaves.

format can all be generated. Figure 5.1 shows a diagram of the code that is generated from the protocol specification.

I begin with a description of the protocol-definition language I embedded into Haskell. Following that, I show how a portion of the USB protocol is defined using this language, specifically the `GET_DESCRIPTOR` request message sent from the device to the host during enumeration. I extrapolate on that definition to describe the code generated for the entire set of messages and close with a description of how I integrated this generated code into the FreeBSD kernel.

Note that the definitions and generated code I describe in this chapter are a result of my efforts related to USB. They are, however, directly relevant to *any* protocol for which one wishes to implement parsers (though, admittedly, they are more amenable to binary protocols like TCP/IP rather than text protocols like SMTP and HTTP). My intent is to provide the foundation upon which myriad other parsers can be built and, most importantly, integrated with their existing stacks, replacing notoriously vulnerable ad-hoc implementations [9]. For this reason, in what follows I describe the construction of my USB parser in most generic terms, only using protocol-specific terminology when such is specific to USB (as in the case of USB enumeration).

5.1 Protocol Definition

I define a Protocol to be a set of Messages. Each Message consists of a name, a set of Fields, and potentially a data stage of variable length. (Figure 5.2 shows the definition of a Message using

```
data Message = Message MessageName [Field] DataLen
type MessageName = String
```

Figure 5.2: Specification of a Message within a protocol. A Message is defined by a name (a String), a list of Fields, and a data length specifier.

the domain-specific language.) As simple as it appears, this definition describes every message exchanged during USB enumeration, dubbed "control messages". Following enumeration, however, most communication is application-specific and supporting such protocols using this framework reduces to the task of creating Message variables corresponding to the application-specific messages. Control messages still flow between device and host even after enumeration is complete, however, as they negotiate features like flow control and isochronous transfers.

The first value in a Message is a character string used for identification purposes. Following the name is a list of Fields that make up the Message (how to specify names, types, etc. for these fields is discussed below). Fields are assumed to be ordered and contiguous within the Message; should the protocol specify empty space or padding, one would need to specify an explicit Field reflecting those characteristics.

Let us express these and further relationships between the elements in a message in a grammar that, at the same time, is the Haskell definition of the types within the protocol DSL. Message and field names are thus Haskell type constructors; the entire DSL is thus a runnable definition and is therefore subject to the Haskell type-checking framework, which is an effective form of static verification [41].

Each Field in a Message consists of a name and a size, as well as an indication of whether its contents are literal or variable. (The definition of the Field type is shown in Figure 5.3.) Like Message, a Field incorporates a character string used to identify it. The size of the field is either 8 or 16 bits—this covers all messages exchanged during USB enumeration and could easily be expanded to support the needs of other protocols, even those requiring bit-granularity.

The last field indicates whether the field is literal or variable. Many protocols specify an exact sequence of bits or bytes to appear in certain places: for instance, IPv4 requires that the first 4 bits of an IP packet be 0100, indicating the version of IP to which the packet conforms. Likewise, USB requires that, e.g., `GET_DESCRIPTOR` request messages have a RequestType field of 0x80 and a Request value of 6. These would be specified as Literal fields, along with the value they require.

Alternatively, some fields are not precisely specified and instead must be available to higher-level

```
data Field = Field FieldName FieldSize FieldValue

type FieldName = String
data FieldSize = Uint8
               | Uint16
data FieldValue = Literal Int
                | Variable
```

Figure 5.3: Specification of a Field within a Message. A Field is defined by a name (a String), a size (in this case, 8 or 16 bits), and an indication of whether its value is variable or fixed

```
data DataLen = NoData
             | Bytes Int
             | Ref FieldName
```

Figure 5.4: Specification of the length of the data stage of a Message. The length can be zero, a fixed number of bytes, or a number of bytes given in one of the fields of the Message.

code that, e.g., changes state within the kernel upon the receipt of such a message. Examples of this include the destination port number in TCP and the index of the descriptor being requested by a USB `GET_DESCRIPTOR` request message. These would be specified as Variable fields so that the appropriate code can be produced.

The list of Fields is of fixed size and the Fields themselves are of fixed size; thus, the entire Message described so far is of fixed size. These fields may be followed by a variable-sized data stage: the size may be zero, a fixed size, or of a size given by one of the fields. The definition of the DataLen type is shown in Figure 5.4.

Many messages in the USB protocol communicate no data, and therefore use the NoData constructor for this field. Some messages include a fixed amount of data following the header, in which case they use the "Bytes" constructor, specifying the number of bytes as the argument. Lastly, some messages (of which the response to the `GET_DESCRIPTOR` request is one) specify the length in one of the Fields, in which case the "Ref" constructor is used. If the length is specified in the `wLength` field, the DataLen constructor would appear as `Ref "wLength"`.

Note that while the DataLen construct fulfills the needs of USB, it also immediately supports protocols like IP and SCSI that have a similar structure. The latter (and many other protocol besides) feature headers comprised of fixed-size fields followed by a variable-sized payload or data field. This syntactic simplicity indicates the data modeling approach is likely generally applicable.

```
getDescriptorRequest :: Message
getDescriptorRequest = Message "GET_DESCRIPTOR req"
    [ Field "request_type"  Uint8  (Literal 0x80)
    , Field "request"       Uint8  (Literal 6)
    , Field "desc_type"     Uint8  Variable
    , Field "desc_index"    Uint8  Variable
    , Field "language_id"   Uint16 Variable
    , Field "desc_length"   Uint16 Variable
    ]
    NoData
```

Figure 5.5: Description of USB protocol's `GET_DESCRIPTOR` request message, written in the domain-specific language.

5.1.1 Example: `GET_DESCRIPTOR` request message

As an example, consider the `GET_DESCRIPTOR` request message, whose specification is shown in Figure 5.5. The first parameter specifies the name (which will be used in a number of places in the generated code). Following that are six fields: four 8-bit unsigned integers and two 16-bit unsigned integers. The first two fields have literal values whereas the final four are marked as variable, to be interpreted above the parsing layer. Finally, this message does not include any trailing data.

The `GET_DESCRIPTOR` message exercises most of the message-specification features discussed above—fields of different sizes, of both literal and variable contents, as well as a null data stage—but not all. The fixed-length data stage is used in the `GET_STATUS` response, `GET_CONFIGURATION` response, and `SYNCH_FRAME` messages. The variable-length data stage is used in a number of messages, many of which are closely related to other messages.

5.1.2 Specifying New Messages from Old Messages

Many protocols include related pairs of messages; think ICMP echo request and reply, DNS request and reply, and so on. The USB protocol does, as well; the `GET_DESCRIPTOR` request message shown above is the request half of such a pair. For our protocol syntax specification to be complete, we need to specify the format of the response, but it seems wasteful and potentially error-prone to specify the message entirely from scratch.

For USB, many of these request/response pairs differ only in that the response includes a data stage and the request does not. Therefore, I created a Haskell function, `withData` that takes a Message instance, gives it a new name and a new data stage specification, and produces a new Message. Figure 5.6 shows how I used this function to specify the `GET_DESCRIPTOR` response message.

```
getDescriptorResponse :: Message
getDescriptorResponse = withData getDescriptorRequest
                                 "GET_DESCRIPTOR response"
                                 (Ref "desc_length")
```

Figure 5.6: Code to derive the `GET_DESCRIPTOR` response from the associated request message, using the domain-specific language.

While the `withData` function is no doubt useful, it is most certainly specific to USB. The general lesson here is not, however, that the framework I've created is inextricably tied to USB; rather, this demonstrates the power of an embedded domain-specific language. Because the protocol-specification language is really just Haskell, we have at our fingertips all the tools that Haskell provides, which let us quickly, easily, and—most importantly—reliably produce specifications of derived messages. Were we left to specify these messages by hand in entirety, we run the risk of introducing typos and inconsistencies, both of which are a breeding ground for vulnerabilities.

5.2 Generating the Code

As shown pictorially in Figure 5.1, we can use the message definitions described in Section 5.1 to produce other code pertaining to those messages. These various code artifacts are described in the following subsections. All generation code is written in Haskell.

5.2.1 Generating the Data Structure

First and foremost, we need a data structure to represent each message. This structure can (and likely should) be used both in the kernel proper as well as the parsing component. Additionally, it could be used in programs that inject protocol data such as the Facedancer and its associated software described in Chapter 3 as well as programs such as `tcpdump` that analyze protocol traces. Figure 5.7 shows the structure definition generated from the `GET_DESCRIPTOR response` message shown in Figure 5.6. (This example shows the response rather than the request because the presence of a data stage in the response exposes a particular implementation quirk that merits mention.)

Defining fields for fixed-size types is straightforward: the `Uint8` and `Uint16` of the definition from Figure 5.5 become `uint8_t` and `uint16_t`, which are types supplied by standard system headers. The only deviation from this pattern is the `data` member.

For the optional data stage, I have chosen to represent it as a single byte in the structure.

```
struct get_descriptor_req_msg {
    uint8_t request_type;
    uint8_t request;
    uint8_t desc_type;
    uint8_t desc_index;
    uint16_t language_id;
    uint16_t desc_length;
    uint8_t data;
};
```

Figure 5.7: Generated C structure for the `GET_DESCRIPTOR` request message.

Should the kernel, application, or parser wish to access the contents of the data stage, they need to do so using the *address of* the `data` member. This is potentially fraught with peril as undisciplined pointer operations are a significant source of security vulnerabilities. Therefore, I have also generated accessor functions (described in Section 5.2.3) that unify the method of access and therefore reduce the potential for misuse.

5.2.2 Generating the Parser/Verifier

The primary purpose of the parser/verifier function is to ensure that the raw bits received over the wire (metaphorical or otherwise) conform to the protocol specification. It must check both the contents of the individual fields where applicable as well as aspects of the entire frame—most significantly, its length, so as to avoid vulnerabilities such as Heartbleed [46]. The generated parser function for the `GET_DESCRIPTOR` request message is shown in Figure 5.8.

Some things in this function are worthy of note. First, many of the fields are *not* examined: this is reasonable because the contents of those fields either do not affect the validity of the message or their validity is only verifiable given more information about the state of the connection. In short, this function is concerned with message syntax, not semantics.

For instance, the `desc_index` field of a `GET_DESCRIPTOR` response message should match the `desc_index` field of the initial `GET_DESCRIPTOR` request, but the parser cannot know such things without maintaining significant application-specific state. Such state is more the purview of a separate component that verifies the validity of *sequence* of messages rather than each individual message in the sequence; this work is focussed solidly on the latter problem. A similar separation exists in the NetFilter architecture, where keeping track of state is relegated to distinct code such as ConnTrack, which keeps track of stateful protocols such as TCP. The state tracked may be exact as per protocol specification or, as in the case of TCP, approximated.

```
struct get_descriptor_req_msg *
validate_get_descriptor_req_msg(char *frame, int framelen)
{
    struct get_descriptor_req_msg *m =
            (struct get_descriptor_req_msg *) frame;

    if(m == NULL) return NULL;
    if(framelen < 1 + 1 + 1 + 1 + 2 + 2 + 0) return NULL;
    if(m->request_type != 128) return NULL;
    if(m->request != 6) return NULL;
    /* accept m->desc_type as-is */
    /* accept m->desc_index as-is */
    /* accept m->language_id as-is */
    /* accept m->desc_length as-is */
    return m;
}
```

Figure 5.8: Generated verification function for the `GET_DESCRIPTOR` request message. (Constant-folding in the compiler will optimize away the tacky addition.)

```
#define get_get_descriptor_req_msg_desc_type(m) (m->desc_type)
#define get_get_descriptor_req_msg_desc_index(m) (m->desc_index)
#define get_get_descriptor_req_msg_language_id(m) (m->language_id)
#define get_get_descriptor_req_msg_desc_length(m) (m->desc_length)
#define get_get_descriptor_req_msg_GET_DESCRIPTOR_data(m) (&m->data)
```

Figure 5.9: Generated C accessors for the `GET_DESCRIPTOR` request message. (The duplicate "get" substring is not a typo: the first a verb, the second is part of the noun.)

5.2.3 Generating Accessor Functions

While the data members of structures generated by the code described in Section 5.2.1 can be used to access the individual fields of a message, there are advantages to using discrete accessor functions, and compiler tricks can make them just as efficient as direct access methods. Figure 5.9 shows the accessor functions for the `GET_DESCRIPTOR` request message.

The usability of these accessor macros could be improved by implementing them as functions instead, which would allow the compiler to provide more meaningful error messages. The type of the parameter `m` would then be specified (whereas in a macro it is not), thus nominally ensuring that only the correct type of message has its accessed in this way. (One could imagine a case where a different kind of message also has a field named `desc_length`, but located in a different place within the message. The macros do not protect against using an instance of the latter in place of the former, whereas a function would.) Such functions should probably be marked as `inline` so

```
void
print_get_descriptor_req_msg(struct get_descriptor_req_msg *m)
{
    log(LOG_INFO, "usb_fw: GET_DESCRIPTOR req, desc_type=%d,"
            "desc_index=%d,language_id=%d, desc_length=%d\n",
            m->desc_type, m->desc_index, m->language_id,
            m->desc_length);
    log(LOG_INFO, "usb_fw: data stage=%s\n", bytes_as_hex(&m->data,
            m->desc_length));
}
```

Figure 5.10: Generated C function for legibly printing a `GET_DESCRIPTOR` request message.

that the compiler can produce code as efficient as if they were macros.

Another advantage of using accessor functions (macros) like these is that any endianness modifications can be incorporated into the functions themselves. While this isn't an issue in USB, it most certainly *is* an issue in traditional networking protocols such as the TCP/IP stack. Using only accessor functions (macros) that have the endianness conversion incorporated could be a benefit.

Admittedly, these names might be unwieldy. The good news is that, being automagically generated, they can be easily changed. For instance, one could write a function to shorten names and apply it to all identifiers simultaneously.

5.2.4 Generating the Pretty-Printer Function

We also require the ability to present the details of a message in a user-friendly format. While perhaps not strictly necessary within the kernel proper, this feature is vital to user-facing tools that inspect protocol traffic. (For example, a protocol-specific tool analogous to `tcpdump` for TCP/IP protocols.) Figure 5.10 shows the generated pretty-printing function for the `GET_DESCRIPTOR` request message.

Note that each field is correctly formatted according to its type, the literal fields are elided from the output, and the data stage is outputted as hex, using the previously-verified length. (NB: the generated function uses the FreeBSD-specific `log` function and `LOG_INFO` log-level. The reasons for this are explained in Section 5.5.2.)

5.3 User-Defined Policies

As described in Section 1.2, one of the features we would like in a firewall is the ability to specify a policy to augment the built-in rules. For instance, imagine a case where a particular USB device

```
Reject string_descriptor where length = 42
Reject set_address where address > 127
```

Figure 5.11: Example user policies for the USB firewall.

driver doesn't correctly handle a string descriptor with a length of exactly 42. Instead of entirely disabling support for that device or waiting for a new driver, a system administrator might want to filter out all USB frames that contain the offending length value.

To fulfill this need, I devised a simple language to describe policies, a parser for that language, and code that uses the previously-written protocol definition to produce a loadable kernel module that implements the policy. Currently, the policy in this kernel module is applied after the frame is validated but before the connection-tracking logic to be described in Section 5.5. (An improvement to my system would be to allow for more flexible policy-application orderings.) Some example policies are shown in Figure 5.11.

These policies are, admittedly, not especially eloquent. In particular, they do not take into account the context in which a message is being sent and are therefore something of a blunt instrument. A subtler and more targeted approach would be to augment a detailed state machine with such rules, but that is more semantic than syntactic and is beyond the scope of this work. Again, however, NetFilter shows a possible direction in which modules specified with the policy provide additional predicates for checking state as needed.

5.4 Protocol: Assemble!

The preceding sections have described the generation of individual chunks of code necessary for each message of the protocol in question. What remains is to generate all these code chunks for every message, place them in well-formed source files, and integrate them with the target operating system.

For the USB protocol proof-of-concept, I defined instances of the `Message` type for the following messages (where applicable, related message types are listed together).

- `GET_STATUS` request response
- `CLEAR_FEATURE` and `SET_FEATURE`
- `SET_ADDRESS`

- `GET_DESCRIPTOR` request
- `SET_DESCRIPTOR`
- `GET_CONFIGURATION` request and response
- `SET_CONFIGURATION`
- `GET_INTERFACE` request and response
- `SET_INTERFACE`
- `SYNCH_FRAME`

I also defined instances of the `Message` type for the following descriptors. These descriptors are sent in the data stage of responses to the `GET_DESCRIPTOR` request message defined above.

- device descriptor
- configuration descriptor
- interface descriptor
- endpoint descriptor
- string descriptor
- hid descriptor
- report descriptor

Taken together, these requests, responses, and descriptors encompass all data that flows between host and device during the USB enumeration process.

The data structure definitions, accessor macros, and function prototypes are generated into a file called `usb_messages.h`. The validation functions and pretty-printing functions are generated into a file called `usb_messages.c`. Both of these source files are intended to integrate with any operating system kernel or application (though a few idiosyncrasies remain: see Section 5.5.2).

5.5 Operation System Integration

This generated code is all well and good, but it needs to get itself into an operating system to make any difference. I achieve this by means of a thin translation shim, described below, whose design was guided by the instrumentation described in Chapter 4.

I chose to integrate with FreeBSD because of its reputation as a widely-deployed, high-performance kernel with a clean and well-documented design. (After considerable time spent digging through kernel source code, I concluded this reputation is merited.)

Running the Haskell code on the USB protocol specification results in two files, `usb_messages.h` and `usb_messages.c`, that implement the various constructs described in this chapter. They are intended to be as operating-system agnostic as possible (exceptions are discussed in Section 5.5.2).

I separately implemented a FreeBSD kernel module that, when loaded, provides a function that the mainline USB stack can call to verify a set of frames. This function is primarily responsible for extracting the relevant fields of the structure FreeBSD uses to describe a USB transfer and calling an OS-agnostic function with the frame and the extracted fields as parameters. The idea is that integrating with a new operating system will require one to re-implement only this *translation shim* and leave the rest of the validation code intact.

This OS-agnostic code is contained in `usb_fw.h` and `usb_fw.c`, which currently implements a simple policy in which a response is verified to match the request that instigated it. It is intended not to demonstrate a complicated, stateful firewall for USB but rather how the primitives provided by the automagically-generated code can be used to do so.

Table 5.1 summarizes the files involved. The primary takeaway from this table is the significant discrepancy between manually-written lines of code and automagically-generated lines of code, the latter of which are far more likely to be correct. This is not because of some magical fairy dust involved in the autogeneration process, but rather because all of the code is produced in a uniform fashion. Bugs need only be fixed once in the generation code and all the constructs that are generated are positively affected. In contrast, fixing a single bug in a manually-written parser does not guarantee that same bug doesn't exist in another component that performs a similar operation.

Once the kernel module is loaded, a frame is processed thusly:

1. When execution reaches one of three points in the USB stack, call `fbsd_hook`, giving it the FreeBSD-specific structure that describes the transfer (which may contain multiple, raw USB frames). In Section 5.5.1, I describe the method I developed to place these hooks.

2. Within `fbsd_hook`, extract transfer metadata—such as bus number, device address, and endpoint number—from the FreeBSD-specific structure and pass each frame in turn to `hook_frame` along with the OS-agnosticized metadata.

3. The `hook_frame` function validates the frame, which results in an action (such as accept, drop, or reject) being passed back to `fbsd_hook`.

Filename	LOC	Description
`usb_messages.h`	403	structure definitions, accessor macros definitions, and function prototypes (auto-generated)
`usb_messages.c`	367	parser functions and pretty-printing functions (auto-generated)
`usb_fw_fbsd.c`	105	FreeBSD-specific code, contains `fbsd_hook` function that invokes OS-agnostic code
`usb_fw.h`	14	definitions for OS-agnostic functions
`usb_fw.c`	71	rudimentary firewall for USB using the auto-generated parser primitives

Table 5.1: The files, both automagically and manually generated, that comprise the USB validation proof-of-concept; along with their sizes, measured in lines of code, and a brief description of their purpose.

4. Finally, `fbsd_hook` returns the action back to the USB stack.

Figure 5.12 shows the path by which a frame is processed by the generated verification framework. But where are these magical, "appropriate places" whence `fbsd_hook` is called?

5.5.1 Hooks

The method for locating hooks is as follows. Although applied to USB in this chapter, it can be easily generalized to other protocols.

In particular, given the `fbsd_hook` function described in the previous section, where in the USB stack proper does it get invoked? The instrumentation described in Chapter 4 revealed that all frames entering the kernel over USB did so in the `usbd_callback_wrapper` function and that all frames exiting the kernel over USB did so in either the `usbd_transfer_start_cb` or `usbd_pipe_start` functions. Therefore, it was in those functions that I placed the hooks to call into the firewall. I evaluate the effectiveness of these placements in the next chapter where I discuss whether my system fulfills the requirement of complete mediation as described in Section 1.4.

5.5.2 Obstacles to Operating System Independence

This is not to say that the idiosyncrasies introduced by particular operating systems are trivial: much depends on the coding style of the operating system in question. These idiosyncrasies for

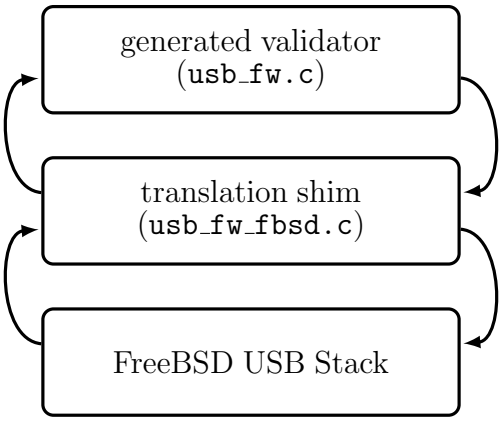

Figure 5.12: When a USB frame arrives or is sent, the FreeBSD USB stack calls the shim function, `fbsd_hook`, which translates the FreeBSD-formatted USB frame metadata to an OS-agnostic format before passing it along to the generated parser/validator function. The resulting action is cascaded back to the kernel.

FreeBSD are described in the following.

The vast majority of the generated code described in the preceding sections is operating-system agnostic; header files are the primary exception. For instance, the `uint8_t` type is used frequently, but the file in which it is defined varies. The FreeBSD kernel uses `<sys/types.h>`, the Linux kernel uses `<linux/types.h>`, and both userlands use `<stdint.h>`. The generated code currently supports only FreeBSD with hard-coded header-file inclusions, but this could easily be expanded to other operating systems either by generating `#ifdef/#endif` clauses for each or by adding an abstraction layer that allows the author to specify differences between platforms.

The other operating-system specific code, as foreshadowed in Section 5.2.4, comprises the functions generated to pretty-print the content of messages. As shown in Figure 5.10, these functions currently use the logging interface exposed by the FreeBSD kernel. There are a few different ways this could be ported to another operating system. One is by using an OS-specific abstraction layer as suggested to solve the header-file problem described in the previous paragraph.

My current preference, however, is to re-implement these functions to instead behave like `snprintf`: returning a pointer to a string instead of perforning the actual logging itself. One benefit of this approach is that such code could be used outside the kernel (e.g., in a program like `tcpdump` that monitors traffic on a bus and presents it in a user-friendly format). The difficulty is that allocating memory for such strings inside the kernel can be a delicate affair, and different kernels prefer different patterns for doing so. Therefore, it seems like an OS-specific abstraction layer is inevitable, but

this merits investigation before committing to a particular strategy.

5.6 Putting It All Together

This chapter presents the key component of my programme: the automagic generator that turns a protocol description written in an embedded domain-specific language into the code ready for integration into a production operating system as well as specific issues related to such integration. I created this code for the USB protocol, and my work to integrate this implementation with the FreeBSD kernel. It now remains to evaluate the resultant system according to the desiderata enumerated in Section 1.2; namely, that the resulting system preserves the protocol functionality and adds the ability to filter out malicious traffic. This is the topic of the next chapter.

5.7 Realization

The mechanisms I outline in this chapter are more involved, complex, and far-reaching than those in previous chapters and therefore the responsibility for implementing them is more distributed. The definition of the formal protocol specification itself is best left to the group creating the reference implementation of the protocol. This group could also implement the code that generates all the operating-system agnostic functions: the message verifier, accessors, pretty-printers, and user-defined policy engine. (Should either of these not happen, one strength of the design is that *any* entity could produce these components and share them with all interested parties.) Then it is up to the maintainers of individual operating systems to implement the shim that bridges their kernel with the generated code.

Finally come the user-defined policies. With support built into the implementations as generated, the question becomes: who writes the policies and who distributes them? The first point to consider is that policies could serve multiple purposes. Some organizations could seek to limit the types of, e.g., USB devices that can be connected to computers under their control, in which case the orgnization's IT department would be responsible for devising and distributing their set of preferred policies. On the other hand, policies that protect against particular vulnerabilites could be written by the discoverer of the vulnerability or the vendor of the vulnerable component. In either case, the policy would be distributed via some public mechanism such as the Web or an e-mail list and the user would be responsible for enabling it on their individual machine.

Chapter 6

Evaluation

The USB firewall described in the previous chapter is all well and good, but how do we know it performs as advertised? That is, how do we know that the generated parser is actually effective? The following questions examine the various angles of *effectiveness* of a firewall.

1. Is the firewall stable? Does it handle the USB protocol without crashing? Inserting more code into the kernel runs the risk of adding points of instability that an attacker could exploit to deny access to the rest of the machine. We must show that the firewall does not crash in the face of both legitimate and malformed USB traffic.

2. Does the firewall examine *every* USB frame received by the computer? If there exist avenues by which frames can reach the kernel innards without being inspected by the firewall, the firewall is not doing its job. We must show that all frames pass through the policy. (This is the notion of *complete mediation*, described in Chapter 1.)

3. In performing its job, does the firewall incur a reasonable amount of overhead? One of the claims made in Chapter 1 is that the generated code would be sufficiently performant to justify its inclusion in a production-quality kernel: we must show that.

4. Finally, and most significantly, does the firewall prevent malformed USB frames from entering the kernel? That is, does the firewall actually do the job it claims to do?

This chapter will answer these questions in the context of the USB firewall I implemented.

Figure 6.1: Diagram of testing setup. Software (in this case, umap) running on the "testing host" (left) causes the Facedancer to emulate a variety of USB devices when connected to the "target" (right).

6.1 Methodology

The primary tool I used to test my USB firewall was umap [21], a USB host security assessment tool designed to test a broad cross-section of USB devices and, by extension, a broad cross-section of the USB protocol itself. Written by Andy Davis of NCC Group, it uses the Facedancer hardware described in Chapter 3 to emulate a wide variety of devices, both well-behaving and otherwise. The umap application itself is built on top of the software stack I wrote and described in Chapter 3. To the best of my knowledge, umap represents the state-of-the-art in testing the security of host-side USB implementations.

The umap test suite contains the largest set of known USB vulnerability triggers. All told, I ran nearly its 500 different vulnerability triggers against my USB parser/firewall; a finer breakdown of the tests is shown in Table 6.1.

Figure 6.1 shows the testing setup. The Facedancer, which umap uses to physically inject its stimuli onto the USB, has two ports: "host" and "target". The former is connected to a USB port on the machine controlling the test and the later is connected to a USB port on the machine being tested. When these connections are made, the host detects a standard USB serial device whereas the target detects no device at all. Only when software (e.g., umap) is run on the host that causes the Facedancer to emulate a particular device does the target actually see a device connect. Once that happens, the software running on the host controls nearly all aspects of the emulated device's behavior (exceptions discussed below).

Using this setup, I first identified the drivers supported by both umap and FreeBSD and then I tested the intersection of those sets.

6.1.1 Identifying testable drivers

The umap software package supports a variety of testing modes. I first ran it in "identification" mode to determine which devices were supported by the FreeBSD target so that I could focus on

```
\$ ./umap.py -P /dev/ttyUSB0 -i
01:01:00 - Audio : Audio control : PR Protocol undefined
 **SUPPORTED**
01:02:00 - Audio : Audio streaming : PR Protocol undefined
 **SUPPORTED**
02:02:01 - CDC Control : Abstract Control Model : AT commands V.250

02:03:ff - CDC Control : Telephone Control Model : Vendor specific

02:06:00 - CDC Control : Ethernet Networking Control Model : No \
  class-specific protocol required
03:00:00 - Human Interface Device : No subclass : None
network socket=False
**SUPPORTED**
06:01:01 - Image : Still image capture device : Bulk-only protocol

07:01:02 - Printer : Default : Bidirectional interface
**SUPPORTED???**

08:06:50 - Mass Storage : SCSI : BBB
**SUPPORTED**
09:00:00 - Hub : Default : Default
**SUPPORTED**
0a:00:00 - CDC Data : Default : Default

0b:00:00 - Smart Card : Default : Default
```

Figure 6.2: Output of umap running in identification mode. (Slightly edited to remove umap banner and long lines.) Of the device classes testable by umap, five are supported in the FreeBSD target: audio control, audio stream, human interface devices (mice and keyboards), printers, mass storage (e.g., thumbdrives), and hubs.

these in the remainder of my testing. Figure 6.2 shows the output of this mode.

First and foremost, this output demonstrates that the kernel on the machine being tested did not ever crash while being probed by umap—*despite umap being a tool explicitly designed to cause such crashes!* This is a first step to showing that the firewall is stable in the face of real USB traffic. Secondly, the output tells us that, of the many device classes supported by umap, six are also supported by FreeBSD and are therefore available to fuzz: audio control, audio streaming, human interface devices (e.g., mice and keyboards), printers, mass storage (e.g., thumbdrives), and hubs.

6.1.2 Fuzz-testing individual drivers

With these six device classes in hand, I proceeded to test each individually using umap's fuzz-testing feature. This feature causes the Facedancer to emulate a particular device and, as part of the USB

enumeration phase, send one frame that pushes the bounds of the specification. For instance, where the kernel might expect to receive an 8-bit field that contains `0x02`, umap would perform one test where it sends `0x00` in this field and another where it sends `0xFF`, the idea being to verify that the kernel safely handles extreme cases.

For each device class, umap supports a large number of such tests: I ran them all. The USB firewall was configured with no user-policy rules; only the generated validation functions were invoked. Figure 6.3 shows a sample of the output from a single fuzz-testing run. Each line represents a single test, the nature of which is described on the far right.

I ran into an interesting issue when using umap to test human interface devices (HID, class 03:00:00 from Figure 6.2). The firewall successfully recognizes and rejects umap's "Configuration_bDescriptorType_null" test, in which the emulated device sends a configuration descriptor with the bDescriptorType field set to 0x00. But because this malformed descriptor is silently rejected, FreeBSD continues to wait for a correct response, eventually timing out. When performed repeatedly, this test causes some state within the FreeBSD kernel to become sufficiently out of whack that *no* HID device will be successfully recognized, whether it conforms to the protocol or not. This suggests there is a bug within the FreeBSD kernel that allows for a denial-of-service when performing incomplete enumeration of HID devices. Further umap tests of the HID device class exhibit this behavior as well, so I elided them from the test suite.

Thus, rather than undermining my methodology, this "failure" in fact highlights a potentially significant flaw in the underlying operating system which relies on rejection by timeout rather than rejection by content. While developing this behavior into a proof-of-concept exploit is beyond the scope of this thesis, the root cause is likely non-trivial. The fact remains, however: my system discovered this bug.

Table 6.1 summarizes the results of the fuzzing runs: all tests over all five remaining device classes, totalling 483 different tests and over 6000 frames sent by umap to the FreeBSD target being tested.

Once again, during all this testing, the firewall stayed stable. This is particularly notable because these tests are actively probing the dark, dirty corners of device behavior. If the firewall does not crash under these circumstances, it is highly unlikely that well-behaved devices will cause it to crash. Therefore, considering that umap is an industry-standard tool for testing the stability of USB implementations, my testing suggests the firewall is stable for production use.

This claim of stability might seem unreasonable in the face of the HID behavior described at the beginning of this section. I contend it is eminently reasonable: the firewall itself *did the correct thing*

```
\$ ./umap.py -P /dev/ttyUSB0 -f 01:01:00:A
Fuzzing:
01:01:00 - Audio : Audio control : PR Protocol undefined
 **SUPPORTED**
Enumeration phase...
2015/11/21 19:40:27 Enumeration phase: 0000 - Device_bLength_null
2015/11/21 19:40:34 Enumeration phase: 0001 - Device_bLength_lower
2015/11/21 19:40:41 Enumeration phase: 0002 - Device_bLength_higher
2015/11/21 19:40:48 Enumeration phase: 0003 - Device_bLength_max
2015/11/21 19:40:56 Enumeration phase: 0004 - \
    Device_bDescriptorType_null
2015/11/21 19:41:06 Enumeration phase: 0005 - \
    Device_bDescriptorType_invalid
2015/11/21 19:41:17 Enumeration phase: 0006 - \
    Device_bMaxPacketSize0_null
2015/11/21 19:41:25 Enumeration phase: 0007 - \
    Device_bMaxPacketSize0_max
2015/11/21 19:41:32 Enumeration phase: 0008 - \
    String_Manufacturer_overflow
2015/11/21 19:41:39 Enumeration phase: 0009 - \
    String_Product_overflow
...
```

Figure 6.3: Sample output from umap fuzzing Audio Control devices. The "A" in the final command-line argument causes umap to run "all" tests. The umap banner has been removed and the output has been truncated (full output runs 119 lines and is summarized in Table 6.1).

USB identifier	device type	tests	frames sent
01:01:00	Audio control	94	1873
01:02:00	Audio streaming	94	1873
07:01:02	Printer	131	1735
08:06:50	Mass storage	101	1506
09:00:00	Hub	63	898
	total	483	6397

Table 6.1: Data sent by umap fuzz-testing.

test	sent	received	missed frames
Audio control	1873	1961	0
Audio streaming	1873	1961	0
Printer	1735	1860	0
Mass Storage	1506	1760	0
Hub	898	955	0

Table 6.2: Complete mediation test results. For each test, shows number of USB frames sent by umap and the number of frames processed by the USB firewall of the machine being tested.

under those circumstances whereas the kernel code being protected failed to do the correct thing upon rejection of the frame. Note that all frames are rejected using the same procedure: the "error" field of the USB transfer structure is set to 1. During USB HID device enumeration, the kernel seems to incorrectly handle this return value; whereas it correctly recovers from all other rejections.

While certainly necessary, this declaration is not sufficient to endorse the firewall entirely. It remains to show that it is effective at protecting against the evils it claims to deter.

6.2 Complete Mediation

The first step in showing that the firewall protects against such evils is to show that it actively examines all the data that flows over the bus; that is, that it implements *complete mediation*. To empirically test whether every single frame sent by umap is evaluated by the USB firewall, I configured umap to print a message whenever it sends a frame and I instrumented and configured the firewall to print a message whenever it evaluates a frame. I configured both to also print the raw bytes of the frame being sent or evaluated. Then I ran the entire fuzz-testing suite described in Section 6.1 and gathered the results shown in Table 6.2.

The first two numerical columns tell a bizarre story: how is the firewall receiving *more* frames than are being sent? The answer lies in the MAXUSB controller chip that sits on the Facedancer board itself, which automatically responds to some USB requests without consulting the software stack. For instance, the Facedancer automatically responds to the `SET_ADDRESS` request and thus such a request/response pair shows up in the kernel logs on the FreeBSD target being tested, but the umap log only shows the request being received.

Since I had logged the raw bytes being sent by umap and received by the firewall, I was able to

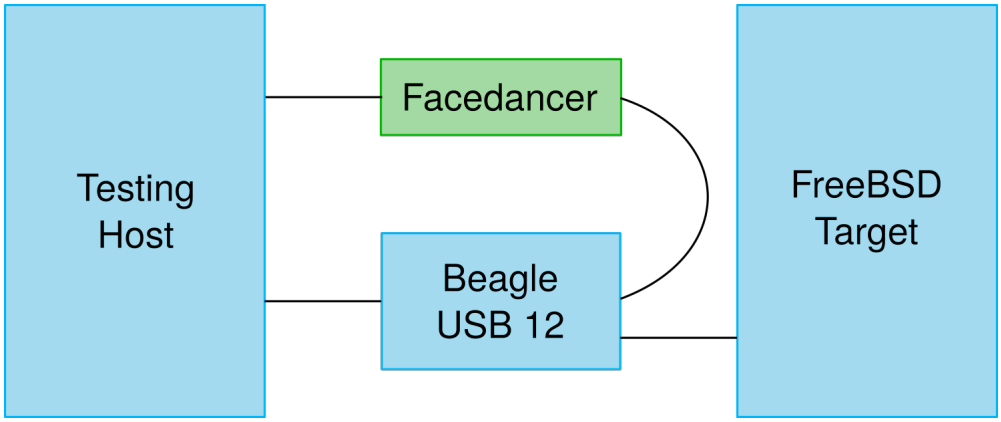

Figure 6.4: To capture USB data sent by the Facedancer, I connected the Beagle USB 12 Protocol Analyzer as a pass-through device between the Facedancer and the FreeBSD target; then I connected the Analysis port of the Beagle back to the testing host to capture packets.

check the differences in actual data being sent and received. *Every single frame sent by umap was analyzed by the firewall.* No exceptions. Some frames were received that the umap software did not send; those all fell into the category of automatic responses generated by the MAXUSB chip on the Facedancer board. But not one frame sent by umap evaded the firewall's oversight.

A More Rigorous Test While these results are encouraging, they are not conclusive. To make them conclusive, I used a Beagle USB 12 Protocol Analyzer [5] from Total Phase to capture all USB data sent by umap to the FreeBSD target. The Beagle USB 12 sits between the Facedancer and the target and mirrors all USB data to the computer connected to its "Analysis" port (see Figure 6.4). I re-ran the umap fuzz tests for the five device classes shown above and recorded all packets observed by the Beagle and all packets mediated by the USB firewall on the FreeBSD target.

As in the informal testing described above, there were some discrepancies between the sequence of USB packets reported by the Beagle and the set of USB packets mediated by the firewall. These discrepancies fell into two categories:

1. When the host (i.e., the FreeBSD target) queries the device (i.e., the Facedancer) and the latter responds with a bare acknowledgement, this acknowledgement appears within the kernel as a message whose contents match the original query. Thus, the firewall will report repeated messages that mirror the preceding message and the Beagle will report empty messages.

2. Some communications span multiple USB packets. These are reported by the Beagle as separate, whereas the USB controller on the FreeBSD host reassembles them before they are passed

CPU	Intel Core i7-4600U
cores	4
clockspeed	2.1 GHz
memory	12 GB

(a) Linux test host.

CPU	Intel Core 2 Duo U7300
cores	2
clockspeed	1.3 GHz
memory	3 GB

(b) FreeBSD test target.

Table 6.3: Specifications of machines used for performance testing (Section 6.3).

to the kernel for processing. Thus, the record of USB messages seen by the Beagle sometimes contains, e.g., three messages whose contents, when concatenated, match the corresponding single message reported by the firewall.

According to my testing, no other discrepancies exist between USB data seen by the Beagle protocol analyzer and the USB firewall. In light of this analysis, my testing suggests that the USB firewall I have integrated into FreeBSD satisfies the complete mediation requirement as laid out at the beginning of this chapter.

6.3 Performance

In addition to being stable and enforcing complete mediation, the firewall must not incur undue performance penalties. To measure the additional processing time induced by the presence of the firewall, I again used the umap fuzz-testing feature. I modified umap to produce as little output as possible and I turned off all logging in the USB firewall. I then ran each test suite three times, rebooting between each test. I used the time(1) program to measure the duration of each fuzz testing run. The specifications of the machines I used for this are shown in Table 6.3; the results for each set of test suite runs are shown in Table 6.4.

These numbers tell a very strange story. For audio control, audio streaming, and mass storage devices, the penalty incurred by activating the firewall is minimal, whereas the effect on printers is moderate and the effect on hubs is significant. Yet it is curious that the disparities are so unevenly spread among device classes; the abysmal performance of the hub class is particularly worrisome.

I investigated this behavior and found that, when the firewall was *disabled*, FreeBSD noticed the erroneous value sent by umap and immediately disconnected the device. By contrast, when I enabled the firewall, the firewall correctly rejected the erroneous frames, but FreeBSD continued to poll the device twice more, with one second between each attempt, until it gave up and disconnected

test	disabled	enabled	impact
Audio control	681 s	690 s	1.3%
Audio streaming	681 s	690 s	1.3%
Printer	459 s	526 s	14.5%
Mass Storage	649 s	668 s	2.9%
Hub	338 s	477 s	41.1%

Table 6.4: Results of USB firewall performance tests. Columns show measured duration of fuzz-testing suite for each device class, averaged over three runs, first with the firewall disabled, then with the firewall enabled, and the measured impact of enabling the firewall as a percentage.

the device. This mirrors the situation I discovered with human-interface devices (described in Section 6.1.2).

It is important to note here that the firewall is doing its job! One could argue that, when the firewall is disabled, FreeBSD is being overzealous in disconnecting the hub immediately on detecting an error. Alternatively, one could argue that this highlights the need for a more nuanced interface between firewall and kernel—that the current firewall is too simplistic in its binary choice of either accept or reject. This work does not attempt to make philosophical judgments along those lines, but further research into the "correct" abstraction to present seems worthwhile. We note that these abstractions come to the forefront due to the integration between the firewall and the underlying operating system—which exposes non-trivial architectural features.

Common-case Performance The previous section describes the minimal overhead incurred when the firewall is presented with invalid traffic. One hopes, however, that most of the traffic examined by the firewall will be benign, therefore I also measured the impact of the firewall on legitimate traffic. I used the `singlestreamread` workload from the FileBench benchmark suite [60] to measure throughput of a USB mass storage device. I ran 5 experiment each with the firewall enabled and disabled. The results are shown in Table 6.5.

Oddly, FileBench reports that performance is *better* when the firewall is enabled compared to when it is *dis*abled. The numbers are so close, however, that there is essentially no difference between the two, and thus I claim that the automagically-generated USB firewall does not incur an unreasonable performance penalty in the face of legitimate USB traffic.

As a final point relative to performance, I should point out that none of the generated code is the

	operations	ops/sec	mb/sec	ms/op
Firewall disabled	32520	542	538	1.8
Firewall enabled	32725	543	542	1.8

Table 6.5: Results of running FileBench's `singlestreamread` workload on a USB mass storage device, with the firewall enabled and disabled.

least bit optimized. It could almost certainly be made far more efficient. Acceptable performance is the goal of this proof-of-concept; exceptional performance can come later.

6.4 Effectiveness

The final criteria by which the effectiveness of the firewall should be judged is whether it successfully prevents "bad" frames from entering the kernel. But what does "bad" mean in this context? Certainly, we wish to prevent frames that deviate from the protocol specification, but there are other circumstances to consider as well. What if the kernel incorrectly handles a frame that is correct according to the specification? Many of the fuzz tests and known-vulnerability tests performed by umap test precisely this possibility. Furthermore, what if a system administrator wants to prevent, e.g., thumbdrives from working on a particular machine?

For example, the USB 2.0 specification decrees that device addresses fall within the range 0–127. Since this is an 8-bit field, a malicious device could conceivably set it to 255 (one of the umap tests does so) and the kernel should reject it without blinking an eye. Detecting this deviation seems outside the realm of the parser because it deals with the contents of the field rather than its boundaries: the specification says the address is an 8-bit field and the parser is (or should be) responsible for taking 8 bits and making a number out of them. Is is *not*, however, out of the realm of a firewall, nor should it be. In fact, it is precisely the purview of the user-policy feature described in Section 5.3. Thus, to enforce the condition that all device addresses fall within the correct range, one could create and enable a user policy including the rule

```
Reject SET_ADDRESS where address > 127
```

This policy is clearly a *mitigation*, not a fix for the particular underlying vulnerability. A fix would eliminate the vulnerability by inserting the check into the operating system code proper; replacing the kernel of a running system, however, has significant operational costs—not to mention the inevitable delay of vendor patch releases. Thus, a mitigation that prevents an exploit payload

from getting to the vulnerable code has great operational value, even though it does not fix the root problem itself. Firewalls were invented as precisely such mitigation tools. They were followed by intrustion-prevention systems (IPS) [2], which continue to evolve to this day.

How, then, can one evaluate the effectiveness of mitigation? While certainly useful, the vulnerability-trigger-based testing enabled by umap and described earlier in this chapter cannot test every possible codepath for the absence of bugs. Since no exhaustive description or model of all bugs is possible, we can only evaluate the effects of a particular mitigation—such as a syntax-based filtering policy—in terms of what we know about the prevalence of bugs in the wild. A useful mitigation should be able to address non-trivial classes of these bugs. That is to say, we need some indication of ground truth with respect to bugs in deployed USB software. For this, we must turn to NIST's National Vulnerability Database (NVD), which catalogs disclosed vulnerabilities as "Common Vulnerability and Exposure" (CVE) records.

My method is as follows. Taking the CVE database as the ground truth of USB vulnerabilities in the wild, I surveyed all reports from the past 10 years (January 2005 through December 2015) that contained the string "usb" and classified them according to their likely relation to errors in parsing syntax. I assume that such vulnerabilities can be mitigated by a syntax-based parser/firewall policy filter while others are unlikely to be so mitigated. My analysis shows that the mitigated class is certainly non-trivial and likely dominant.

Therefore, for each of these 100 vulnerabilities, I reviewed its details and attempted to categorize whether and how the USB protection framework I created could protect against it. This is, admittedly, an imprecise exercise: many of the vulnerability disclosures do not provide sufficient detail to conclusively deduce their cause, which makes it difficult to make substantive claims about them. Even the disclosures relatively devoid of details provide some hints, however. Table 6.6 summarizes the five vulnerability categories I settled on and the vulnerabilities I assigned to each.

6.5 Categories

As I read through the vulnerablity disclosure reports, I assigned each to a category indicating how the USB protection framework described in this dissertation would affect it. I began with a "yes" or "no" classification but, as I proceeded, I was able to produce more nuanced classes, eventually resulting in five different categorizations: unrelated, unclear, mitigated by policy, mitigated by design pattern, and inherently averted.

Appendix A contains tables that list all vulnerabilities in all categories, their summaries, and a

Class	Count	Vulnerabilities
Unrelated	45	CVE-2005-2879 CVE-2005-3055 CVE-2005-4417 CVE-2006-2147 CVE-2006-2936 CVE-2006-6441 CVE-2006-6881 CVE-2007-0734 CVE-2007-0822 CVE-2007-2023 CVE-2007-4785 CVE-2007-5093 CVE-2007-5460 CVE-2008-0708 CVE-2008-0951 CVE-2008-2235 CVE-2008-3150 CVE-2008-3605 CVE-2009-0243 CVE-2009-2834 CVE-2010-0103 CVE-2010-0221 CVE-2010-0222 CVE-2010-0223 CVE-2010-0224 CVE-2010-0225 CVE-2010-0226 CVE-2010-0227 CVE-2010-0228 CVE-2010-0229 CVE-2011-1828 CVE-2012-2693 CVE-2012-6314 CVE-2013-1063 CVE-2013-1774 CVE-2013-3666 CVE-2013-5166 CVE-2014-0860 CVE-2014-2388 CVE-2014-5263 CVE-2014-9596 CVE-2015-1319 CVE-2015-3320 CVE-2015-5960 CVE-2015-6520
Unclear	12	CVE-2005-4788 CVE-2007-3513 CVE-2009-0282 CVE-2010-1140 CVE-2010-3542 CVE-2010-4656 CVE-2011-2295 CVE-2013-0981 CVE-2014-7888 CVE-2014-7893 CVE-2014-7894 CVE-2014-7895
Mitigated by Policy	27	CVE-2005-2388 CVE-2005-4789 CVE-2006-1368 CVE-2007-0728 CVE-2007-6439 CVE-2008-0718 CVE-2009-2807 CVE-2009-2834 CVE-2010-1460 CVE-2010-4530 CVE-2011-0638 CVE-2011-0639 CVE-2011-0640 CVE-2012-4736 CVE-2013-0923 CVE-2013-1860 CVE-2013-2058 CVE-2013-4541 CVE-2013-5192 CVE-2013-5864 CVE-2014-1287 CVE-2014-3185 CVE-2014-3461 CVE-2014-4115 CVE-2015-1769 CVE-2015-5257 CVE-2015-7833
Mitigated by Design Pattern	3	CVE-2010-1083 CVE-2010-3298 CVE-2010-4074
Inherently Averted	14	CVE-2006-2935 CVE-2006-4459 CVE-2006-5972 CVE-2008-4680 CVE-2010-0038 CVE-2010-0297 CVE-2011-0712 CVE-2012-3723 CVE-2012-6053 CVE-2013-1285 CVE-2013-1286 CVE-2013-1287 CVE-2013-3200 CVE-2014-8884

Table 6.6: Classification of USB mentions in CVE incident reports into how they might be affected by the USB firewall I created.

justification for their categorization.

Unrelated Almost half of the vulnerabilities turned up by the search only incidentally touched on USB or didn't relate to data flowing over the bus. For example, CVE-2015-5960 describes an attack whereby a user can bypass Firefox OS permissions and access attached USB mass storage devices. This is not a failure to correctly handle data on the bus, but rather a permissions issue elsewhere in the kernel. Likewise, CVE-2014-5263 describes a failure to correctly terminate a linked list that just happened to be in the USB code. Table A.1 summarizes the vulnerabilities I classified as Unrelated.

Unclear I was unable to categorize about 10% of the USB-related vulnerabilities in my search. CVE-2013-0981, for instance, allows kernel pointers to be modified from userspace, but the disclosure doesn't say whether the userspace application can be affected by traffic from the USB device. And in a shining example of transparency, Oracle declines to specify any details in their vulnerability "disclosures", as exemplified by CVE-2011-2295. Table A.2 summarizes the vulnerabilities I classified as Unclear.

Mitigated by Policy I concluded that nearly one-third of the vulnerabilities could be mitigated by policy. That is, one could write a policy rule that would prevent the USB traffic that exploits the bug. For instance, CVE-2015-7833 is tickled "via a nonzero bInterfaceNumber value in a USB device descriptor"; to prevent such a descriptor from reaching the vulnerable code, one could write a rule that matches device descriptors with a bInterfaceNumber field of zero and, upon a match, rejects the device. Another vulnerability, CVE-2013-5192, is triggered when the USB hub controller in OS X is presented with a request containing a particularly-crafted port number; in this case, one could write a rule that matches and rejects requests with that port number. Table A.3 summarizes the vulnerabilities I classified as Mitigated by Policy.

Mitigated by Design Pattern Three vulnerabilities resulted from deviations from sound programming practices; when sound practices are encoded once in the autogeneration code, such bugs disappear everywhere. Instances include failure to properly initialize structure members (CVE-2010-3298 and CVE-2010-4074) and failure to clear transfer buffers before returning to userspace (CVE-2010-1083). Table A.4 summarizes the vulnerabilities I classified as Mitigated by Design Pattern.

Inherently Averted Finally, almost 15% of the vulnerabilities were due to mistakes in interpreting the structure of the USB messages themselves. Most of these were either buffer overflows that resulted in arbitrary code execution or memory corruption. In both cases, I assumed (dangerously, I know) that some field of the USB descriptor indicating a length did not match the actual length of data provided in the packet. In the generated enforcement code, as long as the original specification of the message is correct, this cannot happen: if one field specifies the length of another, this is verified. Table A.5 summarizes the vulnerabilities I classified as Inherently Averted.

6.6 Results

Of all the vulnerabilities I analyzed, the three categories that bear discussion are those that are inherently averted, those that are mitigated by pattern, and those that are mitigated by policy. I discuss each in turn.

The inherently-averted vulnerabilities are the most straightforward: by clearly defining the structure of messages and dependencies between fields (e.g., that the data stage has a length equal to the value of the "wLength" field) and automagically generating the code to enforce them, an entire class of vulnerabilities can be avoided. The autogeneration code only needs to be audited once and all the generated code can be trusted (especially if it is formally verified), whereas the trustworthiness of manually-written code scattered throughout the kernel is anybody's guess. Furthermore, the declarative nature of the protocol specification language makes auditing much easier than having to dig through procedural code, not least because it more directly matches the form of the published (prose) specification.

My favorite class is the vulnerabilities mitigated by pattern, because here the autogeneration code enshrines good programming practices into the autogeneration framework and thus ensures their proliferation. By causing the autogenerated code to always clear buffers beforehand (for example) then we can depend on buffers to not contain crufty data that might interfere with the computation at hand. Additionally, should new, better practices be developed, we need only incorporate them into the autogeneration framework, regenerate and recompile the code, and suddenly every applicable instance that could be improved, has been improved.

Vulnerabilities mitigated by policy are perhaps the trickiest to appreciate. At first blush, it seems they are not terribly noteworthy: why is being able to write a policy that protects against a vulnerability superior to just fixing the vulnerability itself? The answer lies in practical issues surrounding patching live systems. Distributing and activating a single policy rule to enable protection is much

less disruptive than shipping a newly-compiled binary (still less a *kernel* binary!) containing the fix.

All told, these three classes of vulnerabilities—all of which are addressed, one way or another, by the autogenerated code presented in this dissertation—make up nearly half of all the USB-related vulnerabilities I found in my search, even accounting for the fact that many of the "unrelated" vulnerabilities only coincidentally mentioned USB.

6.7 Summary

This chapter described the results of evaluating my generated parser/firewall in terms of stability, performance costs, ability to mediate malicious traffic, and potential to mitigate USB bugs in the wild (based on the available CVE information). My evaluation, conducted with the state-of-the-art USB security testing suite umap, empirically demonstrates both stability, complete mediation, and reasonable performance for all vulnerability triggers and trigger classes known to date. Additionally, an analysis of available CVE information suggests that large non-trivial classes of bugs in the wild can indeed be mitigated with simple and—importantly—easy to deploy user-defined policies for the firewall. Notably, these mitigations can be deployed immediately and simply on systems which integrate such a firewall, in stark contrast to vendor patches which must be written and tested, and generally require a service interruption to deploy.

Chapter 7

Conclusion

This thesis described the methodology by which I produced a parser/firewall for the USB protocol and integrated it with the FreeBSD operating system, as well as the support software I wrote and explorations I performed leading up to the production of the automagically generated parser/firewall system.

I began with a presentation of the software I wrote to drive the Facedancer board that enables the exploration of USB attack surfaces (Chapter 3). This software mirrors the structure of the USB stack itself and is therefore easier to adapt to new uses, which is a vital feature for exploring attack surfaces. Additionally, the software I wrote was used as the basis for umap, the industry-standard test suite for analyzing the security of USB hosts.

Next, I described the instrumentation framework I applied to the USB subsystem of the FreeBSD kernel (Chapter 4), which allows for observation of a running system at an unprecedented granularity. I showed how I used this framework to measure what percentage of basic blocks were traversed under normal system operation, how I used it to identify where to place enforcement hooks, and how to analyze interactions between software modules.

Then, I presented the methodology I developed to create the USB parser/firewall (Chapter 5). I began with a description of the protocol written in Haskell and automagically generated the data structures, validation functions, printing functions, and accessor functions necessary in a parser for that protocol. I integrated these all into FreeBSD's existing USB stack with the help of a thin translation layer—the idea being that integrating with a different operating system would require only a different translation layer.

I used umap, the comprehensive USB security testing suite, to empirically evaluate the stability,

peformance, and efficacy of my parser/firewall (Chapter 6). I showed that every frame sent by umap was evaluated by the firewall, that the firewall never crashed, and that it was able to handle a variety of user-specified policies. These user policies allow an administrator to respond much more quickly to security issues in the underlying system than the normal process of waiting for and deploying a vendor patch. I showed that, through a combination of beneficial design patterns and user policies, it is likely that a large quantity of known USB bugs are mitigated by my parser/firewall. More importantly, I showed how these features of my parser/firewall mitigate entire classes of bugs as delineated by industry-standard testing tools and processes.

7.1 Future Work

There are a number of directions this work could take from here.

First and foremost, I would like to apply the full seL4-style formal verification process to the autogenerated code. Once that is complete, I would like to work towards getting the code accepted by the FreeBSD kernel maintainers. Additionally, I would like to perform more rigorous testing for complete mediation. (Sadly, the Saturn project [1] for Linux does not have a FreeBSD analog; perhaps that would make a good research project.)

While the Facedancer software framework I wrote is clearly useful—as evidenced by umap built on it—I would like to investigate the feasibility of autogenerating that as well. The basic-block instrumentation I added to the FreeBSD kernel was tedious and screamed for automation: I would like to build such a feature as a compiler plug-in that inserts basic-block instrumentation automatically.

Appendix A

CVE Classifications

This appendix contains the results of my USB-related CVE classification. There is one table for each classification category: unrelated, unclear, mitigated by policy, mitigated by design pattern, and inherently averted. Each table lists the CVE identifier, the official summary, and the reason I chose to classify it as I did.

Rather than provide citations for each individual CVE, the reader is referred to the main portal for the National Vulnerability Database [59], whence once can search for any CVE.

Table A.1: Vulnerabilities classified as Unrelated.

CVE ID	Summary	Justification
CVE-2010-1460	The IBM BladeCenter with Advanced Management Module (AMM) firmware before bpet50g does not properly perform interrupt sharing for USB and iSCSI, which allows remote attackers to cause a denial of service (management module reboot) via TCP packets with malformed application data.	hardware interrupts are below the level of abstraction of the firewall/parser
Cont'd		

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2007-0728	Unspecified vulnerability in Apple Mac OS X 10.3.9 and 10.4 through 10.4.8 creates files insecurely while initializing a USB printer, which allows local users to create or overwrite arbitrary files.	seems unrelated, unless "local user" means "someone who can insert a potentially-malicious USB device", in which case this could likely be averted by policy
CVE-2013-0981	The IOUSBDeviceFamily driver in the USB implementation in the kernel in Apple iOS before 6.1.3 and Apple TV before 5.2.1 accesses pipe object pointers that originated in userspace, which allows local users to gain privileges via crafted code.	seems related to userland code rather than data arriving over USB
CVE-2014-7888	The OLE Point of Sale (OPOS) drivers before 1.13.003 on HP Point of Sale Windows PCs allow remote attackers to execute arbitrary code via vectors involving OPOSMICR.ocx for PUSB Thermal Receipt printers, SerialUSB Thermal Receipt printers, Hybrid POS printers with MICR, Value PUSB Receipt printers, and Value Serial/USB Receipt printers, aka ZDI-CAN-2512.	seems related to other kernel drivers rather than data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2014-7893	The OLE Point of Sale (OPOS) drivers before 1.13.003 on HP Point of Sale Windows PCs allow remote attackers to execute arbitrary code via vectors involving OPOSCheckScanner.ocx for PUSB Thermal Receipt printers, SerialUSB Thermal Receipt printers, Hybrid POS printers with MICR, Value PUSB Receipt printers, and Value Serial/USB Receipt printers, aka ZDI-CAN-2507.	seems related to other kernel drivers rather than data arriving over USB
CVE-2014-7894	The OLE Point of Sale (OPOS) drivers before 1.13.003 on HP Point of Sale Windows PCs allow remote attackers to execute arbitrary code via vectors involving OPOSPOSPrinter.ocx for PUSB Thermal Receipt printers, SerialUSB Thermal Receipt printers, Hybrid POS printers with MICR, Value PUSB Receipt printers, and Value Serial/USB Receipt printers, aka ZDI-CAN-2506.	seems related to other kernel drivers rather than data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2014-7895	The OLE Point of Sale (OPOS) drivers before 1.13.003 on HP Point of Sale Windows PCs allow remote attackers to execute arbitrary code via vectors involving OPOSCashDrawer.ocx for PUSB Thermal Receipt printers, SerialUSB Thermal Receipt printers, Hybrid POS printers with MICR, Value PUSB Receipt printers, Value Serial/USB Receipt printers, and USB Standard Duty cash drawers, aka ZDI-CAN-2505.	seems related to other kernel drivers rather than data arriving over USB
CVE-2005-2879	Advansysperu Software USB Lock AutoProtect (AP) 1.5 uses a weak encryption scheme to encrypt passwords, which allows local users to gain sensitive information and bypass USB interface protection.	encryption is orthogonal to parsing
CVE-2005-3055	Linux kernel 2.6.8 to 2.6.14-rc2 allows local users to cause a denial of service (kernel OOPS) via a userspace process that issues a USB Request Block (URB) to a USB device and terminates before the URB is finished, which leads to a stale pointer reference.	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2005-4417	The default configuration of Widcomm Bluetooth for Windows (BTW) 4.0.1.1500 and earlier, as installed on Belkin Bluetooth Software 1.4.2 Build 10 and ANYCOM Blue USB-130-250 Software 4.0.1.1500, and possibly other devices, sets null Authentication and Authorization values, which allows remote attackers to send arbitrary audio and possibly eavesdrop using the microphone via the Hands Free Audio Gateway and Headset profile.	unrelated to data arriving over USB
CVE-2006-2147	resmgrd in resmgr for SUSE Linux and other distributions does not properly handle when access to a USB device is granted by using "usb:<bus>,<dev>" notation, which grants access to all USB devices and allows local users to bypass intended restrictions. NOTE: this is a different vulnerability than CVE-2005-4788.	unrelated to data arriving over USB
CVE-2006-2936	The ftdi_sio driver (usb/serial/ftdi_sio.c) in Linux kernel 2.6.x up to 2.6.17, and possibly later versions, allows local users to cause a denial of service (memory consumption) by writing more data to the serial port than the hardware can handle, which causes the data to be queued.	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2006-6441	Xerox WorkCentre and WorkCentre Pro before 12.050.03.000, 13.x before 13.050.03.000, and 14.x before 14.050.03.000 allows local users to bypass security controls and boot Alchemy via certain alternate boot media, as demonstrated by a USB thumb drive.	system misconfiguration is unrelated to data arriving over USB
CVE-2007-0822	umount, when running with the Linux 2.6.15 kernel on Slackware Linux 10.2, allows local users to trigger a NULL dereference and application crash by invoking the program with a pathname for a USB pen drive that was mounted and then physically removed, which might allow the users to obtain sensitive information, including core file contents.	unrelated to data arriving over USB
CVE-2007-0734	fsck, as used by the AirPort Disk feature of the AirPort Extreme Base Station with 802.11n before Firmware Update 7.1, and by Apple Mac OS X 10.3.9 through 10.4.9, does not properly enforce password protection of a USB hard drive, which allows context-dependent attackers to list arbitrary directories or execute arbitrary code, resulting from memory corruption.	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2007-2023	USB20.dll in Secustick USB flash drive decouples the authorization and file access routines, which allows local users to bypass authentication requirements by altering the return value of the VerifyPassWord function.	unrelated to data arriving over USB
CVE-2007-5460	Microsoft ActiveSync 4.1, as used in Windows Mobile 5.0, uses weak encryption (XOR obfuscation with a fixed key) when sending the user's PIN/Password over the USB connection from the host to the device, which might make it easier for attackers to decode a PIN/Password obtained by (1) sniffing or (2) spoofing the docking process.	encryption is orthogonal to parsing
CVE-2015-6520	IPPUSBXD before 1.22 listens on all interfaces, which allows remote attackers to obtain access to USB connected printers via a direct request.	vulnerability is in IP-related driver, not USB
CVE-2015-1319	The Unity Settings Daemon before 14.04.0+14.04.20150825-0ubuntu2 and 15.04.x before 15.04.1+15.04.20150408-0ubuntu1.2 does not properly detect if the screen is locked, which allows physically proximate attackers to mount removable media while the screen is locked as demonstrated by inserting a USB thumb drive.	USB used for POC, vulnerability not related to USB itself

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2015-5960	Mozilla Firefox OS before 2.2 allows physically proximate attackers to bypass the pass-code protection mechanism and access USB Mass Storage (UMS) media volumes by using the USB interface for a mount operation.	unrelated to data arriving over USB
CVE-2015-3320	Lenovo USB Enhanced Performance Keyboard software before 2.0.2.2 includes active debugging code in SKHOOKS.DLL, which allows local users to obtain keypress information by accessing debug output.	unrelated to data arriving over USB
CVE-2014-9596	Panasonic Arbitrator Back-End Server (BES) MK 2.0 VPU before 9.3.1 build 4.08.003.0, when USB Wi-Fi or Direct LAN is enabled, and MK 3.0 VPU before 9.3.1 build 5.06.000.0, when Embedded Wi-Fi or Direct LAN is enabled, does not use encryption, which allows remote attackers to obtain sensitive information by sniffing the network for client-server traffic, as demonstrated by Active Directory credential information.	encryption is orthogonal to parsing
CVE-2013-5166	The Bluetooth USB host controller in Apple Mac OS X before 10.9 prematurely deletes interfaces, which allows local users to cause a denial of service (system crash) via a crafted application.	"crafted application" implies userland, not USB device

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2012-4736	The Device Encryption Client component in Sophos SafeGuard Enterprise 6.0, when a volume-based encryption policy is enabled in conjunction with a user-defined key, does not properly block use of exFAT USB flash drives, which makes it easier for local users to bypass intended access restrictions and copy sensitive information to a drive via multiple removal and reattach operations.	unrelated to data arriving over USB
CVE-2013-4541	The usb_device_post_load function in hw/usb/bus.c in QEMU before 1.7.2 might allow remote attackers to execute arbitrary code via a crafted savevm image, related to a negative setup_len or setup_index value.	unrelated to data arriving over USB
CVE-2014-3461	hw/usb/bus.c in QEMU 1.6.2 allows remote attackers to execute arbitrary code via crafted savevm data, which triggers a heap-based buffer overflow, related to "USB post load checks."	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2007-5093	The disconnect method in the Philips USB Webcam (pwc) driver in Linux kernel 2.6.x before 2.6.22.6 "relies on user space to close the device," which allows user-assisted local attackers to cause a denial of service (USB subsystem hang and CPU consumption in khubd) by not closing the device after the disconnect is invoked. NOTE: this rarely crosses privilege boundaries, unless the attacker can convince the victim to unplug the affected device.	unrelated to data arriving over USB
CVE-2008-2235	OpenSC before 0.11.5 uses weak permissions (ADMIN file control information of 00) for the 5015 directory on smart cards and USB crypto tokens running Siemens CardOS M4, which allows physically proximate attackers to change the PIN.	filesystem permissions are orthogonal to parsing
CVE-2008-3605	Unspecified vulnerability in McAfee Encrypted USB Manager 3.1.0.0, when the Re-use Threshold for passwords is nonzero, allows remote attackers to conduct offline brute force attacks via unknown vectors.	user password reuse policy is orthogonal to parsing

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2010-0221	Kingston DataTraveler BlackBox (DTBB), DataTraveler Secure Privacy Edition (DTSP), and DataTraveler Elite Privacy Edition (DTEP) USB flash drives validate passwords with a program running on the host computer rather than the device hardware, which allows physically proximate attackers to access the cleartext drive contents via a modified program.	user password validation is orthogonal to parsing
CVE-2010-0222	Kingston DataTraveler BlackBox (DTBB), DataTraveler Secure Privacy Edition (DTSP), and DataTraveler Elite Privacy Edition (DTEP) USB flash drives use a fixed 256-bit key for obtaining access to the cleartext drive contents, which makes it easier for physically proximate attackers to read or modify data by determining and providing this key.	encryption is orthogonal to parsing
CVE-2010-0223	Kingston DataTraveler BlackBox (DTBB), DataTraveler Secure Privacy Edition (DTSP), and DataTraveler Elite Privacy Edition (DTEP) USB flash drives do not prevent password replay attacks, which allows physically proximate attackers to access the cleartext drive contents by providing a key that was captured in a USB data stream at an earlier time.	authentication is orthogonal to parsing (though a user policy might be able to thwart this particular attack)

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2010-0224	SanDisk Cruzer Enterprise USB flash drives validate passwords with a program running on the host computer rather than the device hardware, which allows physically proximate attackers to access the cleartext drive contents via a modified program.	unrelated to data arriving over USB
CVE-2010-0225	SanDisk Cruzer Enterprise USB flash drives use a fixed 256-bit key for obtaining access to the cleartext drive contents, which makes it easier for physically proximate attackers to read or modify data by determining and providing this key.	poor encryption implementations are orthogonal to USB
CVE-2010-0226	SanDisk Cruzer Enterprise USB flash drives do not prevent password replay attacks, which allows physically proximate attackers to access the cleartext drive contents by providing a key that was captured in a USB data stream at an earlier time.	authentication is orthogonal to parsing (though a user policy might be able to thwart this particular attack)
CVE-2008-3150	Directory traversal vulnerability in index.php in Neutrino Atomic Edition 0.8.4 allows remote attackers to read and modify files, as demonstrated by manipulating data/sess.php in (1) usb and (2) del_pag actions. NOTE: this can be leveraged for code execution by performing an upload that bypasses the intended access restrictions that were implemented in sess.php.	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2010-0227	Verbatim Corporate Secure and Corporate Secure FIPS Edition USB flash drives validate passwords with a program running on the host computer rather than the device hardware, which allows physically proximate attackers to access the cleartext drive contents via a modified program.	user authentication is orthogonal to USB
CVE-2010-0228	Verbatim Corporate Secure and Corporate Secure FIPS Edition USB flash drives use a fixed 256-bit key for obtaining access to the cleartext drive contents, which makes it easier for physically proximate attackers to read or modify data by determining and providing this key.	encryption is orthogonal to USB
CVE-2010-0229	Verbatim Corporate Secure and Corporate Secure FIPS Edition USB flash drives do not prevent password replay attacks, which allows physically proximate attackers to access the cleartext drive contents by providing a key that was captured in a USB data stream at an earlier time.	authentication is orthogonal to USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2014-2388	The Storage and Access service in BlackBerry OS 10.x before 10.2.1.1925 on Q5, Q10, Z10, and Z30 devices does not enforce the password requirement for SMB filesystem access, which allows context-dependent attackers to read arbitrary files via (1) a session over a Wi-Fi network or (2) a session over a USB connection in Development Mode.	user authentication is orthogonal to USB
CVE-2012-6314	Citrix XenDesktop Virtual Desktop Agent (VDA) 5.6.x before 5.6.200, when making changes to the server-side policy that control USB redirection, does not propagate changes to the VDA, which allows authenticated users to retain access to the USB device.	unrelated to data arriving over USB
CVE-2011-1828	usb-creator-helper in usb-creator before 0.2.28.3 does not enforce intended PolicyKit restrictions, which allows local users to perform arbitrary unmount operations via the UnmountFile method in a dbus-send command.	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2012-2693	libvirt, possibly before 0.9.12, does not properly assign USB devices to virtual machines when multiple devices have the same vendor and product ID, which might cause the wrong device to be associated with a guest and might allow local users to access unintended USB devices.	unrelated to syntax of USB messages (and USB stack by design can't know enough to handle this)
CVE-2013-3666	The LG Hidden Menu component for Android on the LG Optimus G E973 allows physically proximate attackers to execute arbitrary commands by entering USB Debugging mode, using Android Debug Bridge (adb) to establish a USB connection, dialing 3845#*973#, modifying the WLAN Test Wi-Fi Ping Test/User Command tcpdump command string, and pressing the CANCEL button.	unrelated to data arriving over USB
CVE-2013-1063	usb-creator 0.2.47 before 0.2.47.1, 0.2.40 before 0.2.40ubuntu2, and 0.2.38 before 0.2.38.2 does not properly use D-Bus for communication with a polkit authority, which allows local users to bypass intended access restrictions by leveraging a PolkitUnixProcess PolkitSubject race condition via a (1) setuid process or (2) pkexec process, a related issue to CVE-2013-4288.	unrelated to data arriving over USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification
CVE-2013-1774	The chase_port function in drivers/usb/serial/io_ti.c in the Linux kernel before 3.7.4 allows local users to cause a denial of service (NULL pointer dereference and system crash) via an attempted /dev/ttyUSB read or write operation on a disconnected Edgeport USB serial converter.	possibly preventable by design pattern, but otherwise an issue of kernel logic unrelated to the USB protocol itself
CVE-2010-0103	UsbCharger.dll in the Energizer DUO USB battery charger software contains a backdoor that is implemented through the Arucer.dll file in the %WINDIR%\system32 directory, which allows remote attackers to download arbitrary programs onto a Windows PC, and execute these programs, via a request to TCP port 7777.	unrelated to data arriving over USB
CVE-2014-0860	The firmware before 3.66E in IBM BladeCenter Advanced Management Module (AMM), the firmware before 1.43 in IBM Integrated Management Module (IMM), and the firmware before 4.15 in IBM Integrated Management Module II (IMM2) contains cleartext IPMI credentials, which allows attackers to execute arbitrary IPMI commands, and consequently establish a blade remote-control session, by leveraging access to (1) the chassis internal network or (2) the Ethernet-over-USB interface.	poor password security is orthogonal to USB

Cont'd

Table A.1 – *Cont'd*

CVE ID	Summary	Justification

Table A.2: Vulnerabilities classified as Unclear.

CVE ID	Summary	Justification
CVE-2008-0718	Unspecified vulnerability in the USB Mouse STREAMS module (usbms) in Sun Solaris 9 and 10, when 64-bit mode is enabled, allows local users to cause a denial of service (panic) via unspecified vectors.	not enough detail
CVE-2009-2807	Heap-based buffer overflow in the USB backend in CUPS in Apple Mac OS X 10.5.8 allows local users to gain privileges via unspecified vectors.	not enough detail
CVE-2013-0923	The USB Apps API in Google Chrome before 26.0.1410.43 allows remote attackers to cause a denial of service (memory corruption) via unspecified vectors.	not enough detail
CVE-2013-5864	Unspecified vulnerability in Oracle Solaris 10 and 11.1 allows local users to affect availability via vectors related to USB hub driver.	not enough detail
Cont'd		

Table A.2 – *Cont'd*

CVE ID	Summary	Justification
CVE-2013-2058	The host_start function in drivers/usb/chipidea/host.c in the Linux kernel before 3.7.4 does not properly support a certain non-streaming option, which allows local users to cause a denial of service (system crash) by sending a large amount of network traffic through a USB/Ethernet adapter.	firewall could potentially perform rate-limiting, but does not currently
CVE-2014-3185	Multiple buffer overflows in the command_port_read_callback function in drivers/usb/serial/whiteheat.c in the Whiteheat USB Serial Driver in the Linux kernel before 3.16.2 allow physically proximate attackers to execute arbitrary code or cause a denial of service (memory corruption and system crash) via a crafted device that provides a large amount of (1) EHCI or (2) XHCI data associated with a bulk response.	firewall could potentially perform rate-limiting, but does not currently
CVE-2005-4788	resmgr in SUSE Linux 9.2 and 9.3, and possibly other distributions, allows local users to bypass access control rules for USB devices via "alternate syntax for specifying USB devices."	not sure if caused by data from USB device or not

Cont'd

Table A.2 – *Cont'd*

CVE ID	Summary	Justification
CVE-2007-3513	The lcd_write function in drivers/usb/misc/usblcd.c in the Linux kernel before 2.6.22-rc7 does not limit the amount of memory used by a caller, which allows local users to cause a denial of service (memory consumption).	not sure if caused by data from USB device or not
CVE-2009-0282	Integer overflow in Ralink Technology USB wireless adapter (RT73) 3.08 for Windows, and other wireless card drivers including rt2400, rt2500, rt2570, and rt61, allows remote attackers to cause a denial of service (crash) and possibly execute arbitrary code via a Probe Request packet with a long SSID, possibly related to an integer signedness error.	not sure if caused by data at USB protocol level or application level
CVE-2010-1140	The USB service in VMware Workstation 7.0 before 7.0.1 build 227600 and VMware Player 3.0 before 3.0.1 build 227600 on Windows might allow host OS users to gain privileges by placing a Trojan horse program at an unspecified location on the host OS disk.	not sure if triggered by USB data
CVE-2010-3542	Unspecified vulnerability in Oracle Solaris 8, 9, and 10, and OpenSolaris, allows local users to affect confidentiality, related to USB.	not enough detail

Cont'd

Table A.2 – *Cont'd*

CVE ID	Summary	Justification
CVE-2011-2295	Unspecified vulnerability in Oracle Solaris 8, 9, 10, and 11 Express allows local users to affect availability, related to Driver/USB.	not enough detail
CVE-2007-6439	Wireshark (formerly Ethereal) 0.99.6 allows remote attackers to cause a denial of service (infinite or large loop) via the (1) IPv6 or (2) USB dissector, which can trigger resource consumption or a crash. NOTE: this identifier originally included Firebird/Interbase, but it is already covered by CVE-2007-6116. The DCP ETSI issue is already covered by CVE-2007-6119.	seems descriptor-related, but not enough information

Table A.3: Vulnerabilities classified as Mitigated By Policy.

CVE ID	Summary	Justification
CVE-2006-4459	Integer overflow in AnywhereUSB/5 1.80.00 allows local users to cause a denial of service (crash) via a 1 byte header size specified in the USB string descriptor.	user policy: "reject message where length ¿ x"
Cont'd		

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2012-3723	Apple Mac OS X before 10.7.5 does not properly handle the bNbrPorts field of a USB hub descriptor, which allows physically proximate attackers to execute arbitrary code or cause a denial of service (memory corruption and system crash) by attaching a USB device.	user policy: "reject message where bNbrPorts == bad value"
CVE-2012-6053	epan/dissectors/packet-usb.c in the USB dissector in Wireshark 1.6.x before 1.6.12 and 1.8.x before 1.8.4 relies on a length field to calculate an offset value, which allows remote attackers to cause a denial of service (infinite loop) via a zero value for this field.	user policy: "reject message where length == 0"
CVE-2005-4789	resmgr in SUSE Linux 9.2 and 9.3, and possibly other distributions, does not properly enforce class-specific exclude rules in some situations, which allows local users to bypass intended access restrictions for USB devices that set their class ID at the interface level.	user policy: "Reject interface_descriptor where class_id = xyz"

Cont'd

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2010-4530	Signedness error in ccid_serial.c in libccid in the USB Chip/Smart Card Interface Devices (CCID) driver, as used in pcscd in PCSC-Lite 1.5.3 and possibly other products, allows physically proximate attackers to execute arbitrary code via a smart card with a crafted serial number that causes a negative value to be used in a memcpy operation, which triggers a buffer overflow. NOTE: some sources refer to this issue as an integer overflow.	user policy: "Reject device_descriptor where serial_number = xyz"
CVE-2011-0638	Microsoft Windows does not properly warn the user before enabling additional Human Interface Device (HID) functionality over USB, which allows user-assisted attackers to execute arbitrary programs via crafted USB data, as demonstrated by keyboard and mouse data sent by malware on a smartphone that the user connected to the computer.	user policy: "reject message where data contains abc"

Cont'd

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2011-0639	Apple Mac OS X does not properly warn the user before enabling additional Human Interface Device (HID) functionality over USB, which allows user-assisted attackers to execute arbitrary programs via crafted USB data, as demonstrated by keyboard and mouse data sent by malware on a smartphone that the user connected to the computer.	user policy: "reject message where data contains abc"
CVE-2011-0640	The default configuration of udev on Linux does not warn the user before enabling additional Human Interface Device (HID) functionality over USB, which allows user-assisted attackers to execute arbitrary programs via crafted USB data, as demonstrated by keyboard and mouse data sent by malware on a smartphone that the user connected to the computer.	user policy: "reject message where data contains abc"
CVE-2013-1860	Heap-based buffer overflow in the wdm_in_callback function in drivers/usb/class/cdc-wdm.c in the Linux kernel before 3.8.4 allows physically proximate attackers to cause a denial of service (system crash) or possibly execute arbitrary code via a crafted cdc-wdm USB device.	not enough detail, but likely user policy to prevent malicious messages from passing into kernel

Cont'd

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2013-5192	The USB hub controller in Apple Mac OS X before 10.9 allows local users to cause a denial of service (system crash) via a request with a crafted (1) port or (2) port number.	user policy: "reject message where port == abc"
CVE-2014-1287	USB Host in Apple iOS before 7.1 and Apple TV before 6.1 allows physically proximate attackers to execute arbitrary code or cause a denial of service (memory corruption) via crafted USB messages.	user policy to reject particularly-crafted messages
CVE-2015-5257	drivers/usb/serial/whiteheat.c in the Linux kernel before 4.2.4 allows physically proximate attackers to cause a denial of service (NULL pointer dereference and OOPS) or possibly have unspecified other impact via a crafted USB device.	user policy to reject particularly-crafted messages
CVE-2007-4785	Sony Micro Vault Fingerprint Access Software, as distributed with Sony Micro Vault USM-F USB flash drives, installs a driver that hides a directory under %WINDIR%, which might allow remote attackers to bypass malware detection by placing files in this directory.	user policy: "reject device_descriptor where product_id == 123"

Cont'd

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2008-0708	HP USB 2.0 Floppy Drive Key product options (1) 442084-B21 and (2) 442085-B21 for certain HP ProLiant servers contain the (a) W32.Fakerecy and (b) W32.SillyFDC worms, which might be launched if the server does not have up-to-date detection.	user policy: "reject device_descriptor where product_id == 123"
CVE-2009-2834	IOKit in Apple Mac OS X before 10.6.2 allows local users to modify the firmware of a (1) USB or (2) Bluetooth keyboard via unspecified vectors.	user policy to prevent offending outgoing message (though firewall does not currently filter outgoing traffic)
CVE-2014-4115	fastfat.sys (aka the FASTFAT driver) in the kernel-mode drivers in Microsoft Windows Server 2003 SP2, Vista SP2, and Server 2008 SP2 does not properly allocate memory, which allows physically proximate attackers to execute arbitrary code or cause a denial of service (reserved-memory write) by connecting a crafted USB device, aka "Microsoft Windows Disk Partition Driver Elevation of Privilege Vulnerability."	user policy to reject particularly-crafted messages
CVE-2015-7833	The usbvision driver in the Linux kernel package 3.10.0-123.20.1.el7 through 3.10.0-229.14.1.el7 in Red Hat Enterprise Linux (RHEL) 7.1 allows physically proximate attackers to cause a denial of service (panic) via a nonzero bInterfaceNumber value in a USB device descriptor.	user policy: "reject device_descriptor where bInterfaceNumber == 123"

Cont'd

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2008-0951	Microsoft Windows Vista does not properly enforce the NoDriveTypeAutoRun registry value, which allows user-assisted remote attackers, and possibly physically proximate attackers, to execute arbitrary code by inserting a (1) CD-ROM device or (2) U3-enabled USB device containing a filesystem with an Autorun.inf file, and possibly other vectors related to (a) AutoRun and (b) AutoPlay actions.	user policy to prevent certain file requests
CVE-2015-1769	Mount Manager in Microsoft Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows 8.1, Windows Server 2012 Gold and R2, Windows RT Gold and 8.1, and Windows 10 mishandles symlinks, which allows physically proximate attackers to execute arbitrary code by connecting a crafted USB device, aka "Mount Manager Elevation of Privilege Vulnerability."	user policy to prevent certain file requests

Cont'd

Table A.3 – *Cont'd*

CVE ID	Summary	Justification
CVE-2009-0243	Microsoft Windows does not properly enforce the Autorun and NoDriveTypeAutoRun registry values, which allows physically proximate attackers to execute arbitrary code by (1) inserting CD-ROM media, (2) inserting DVD media, (3) connecting a USB device, and (4) connecting a Firewire device; (5) allows user-assisted remote attackers to execute arbitrary code by mapping a network drive; and allows user-assisted attackers to execute arbitrary code by clicking on (6) an icon under My Computer\Devices with Removable Storage and (7) an option in an AutoPlay dialog, related to the Autorun.inf file. NOTE: vectors 1 and 3 on Vista are already covered by CVE-2008-0951.	user policy to prevent certain file requests
CVE-2010-4656	The iowarrior_write function in drivers/usb/misc/iowarrior.c in the Linux kernel before 2.6.37 does not properly allocate memory, which might allow local users to trigger a heap-based buffer overflow, and consequently cause a denial of service or gain privileges, via a long report.	user policy: "reject report_descriptor where length ¿ x" or possibly mitigated by design pattern, in which memory allocations are made in a principled fashion

Table A.4: Vulnerabilities classified as Mitigated By Pattern.

CVE ID	Summary	Justification
CVE-2010-1083	The processcompl_compat function in drivers/usb/core/devio.c in Linux kernel 2.6.x through 2.6.32, and possibly other versions, does not clear the transfer buffer before returning to userspace when a USB command fails, which might make it easier for physically proximate attackers to obtain sensitive information (kernel memory).	principled buffer use would be encoded into autogeneration
CVE-2010-4074	The USB subsystem in the Linux kernel before 2.6.36-rc5 does not properly initialize certain structure members, which allows local users to obtain potentially sensitive information from kernel stack memory via vectors related to TIOCGICOUNT ioctl calls, and the (1) mos7720_ioctl function in drivers/usb/serial/mos7720.c and (2) mos7840_ioctl function in drivers/usb/serial/mos7840.c.	principled buffer use would be encoded into autogeneration
CVE-2010-3298	The hso_get_count function in drivers/net/usb/hso.c in the Linux kernel before 2.6.36-rc5 does not properly initialize a certain structure member, which allows local users to obtain potentially sensitive information from kernel stack memory via a TIOCGICOUNT ioctl call.	principled buffer use would be encoded into autogeneration
Cont'd		

Table A.4 – *Cont'd*

CVE ID	Summary	Justification
CVE-2014-5263	vmstate_xhci_event in hw/usb/hcd-xhci.c in QEMU 1.6.0 does not terminate the list with the VMSTATE_END_OF_LIST macro, which allows attackers to cause a denial of service (out-of-bounds access, infinite loop, and memory corruption) and possibly gain privileges via unspecified vectors.	principled data structure use would be encoded into auto-generation

Table A.5: Vulnerabilities classified as Inherently Averted.

CVE ID	Summary	Justification
CVE-2006-2935	The dvd_read_bca function in the DVD handling code in drivers/cdrom/cdrom.c in Linux kernel 2.2.16, and later versions, assigns the wrong value to a length variable, which allows local users to execute arbitrary code via a crafted USB Storage device that triggers a buffer overflow.	length variables are encoded as strictly dependent on other values
CVE-2006-5972	Stack-based buffer overflow in WG111v2.SYS in NetGear WG111v2 wireless adapter (USB) allows remote attackers to execute arbitrary code via a long 802.11 beacon request.	frame lengths automatically enforced given frame specification
Cont'd		

Table A.5 – *Cont'd*

CVE ID	Summary	Justification
CVE-2008-4680	packet-usb.c in the USB dissector in Wireshark 0.99.7 through 1.0.3 allows remote attackers to cause a denial of service (application crash or abort) via a malformed USB Request Block (URB).	malformed data is automatically rejected
CVE-2010-0038	Recovery Mode in Apple iPhone OS 1.0 through 3.1.2, and iPhone OS for iPod touch 1.1 through 3.1.2, allows physically proximate attackers to bypass device locking, and read or modify arbitrary data, via a USB control message that triggers memory corruption.	likely length-related and therefore likely prevented by checking packet length, which is built in
CVE-2010-0297	Buffer overflow in the usb_host_handle_control function in the USB passthrough handling implementation in usb-linux.c in QEMU before 0.11.1 allows guest OS users to cause a denial of service (guest OS crash or hang) or possibly execute arbitrary code on the host OS via a crafted USB packet.	likely length-related and therefore likely prevented by checking packet length, which is built in

Cont'd

Table A.5 – *Cont'd*

CVE ID	Summary	Justification
CVE-2011-0712	Multiple buffer overflows in the caiaq Native Instruments USB audio functionality in the Linux kernel before 2.6.38-rc4-next-20110215 might allow attackers to cause a denial of service or possibly have unspecified other impact via a long USB device name, related to (1) the snd_usb_caiaq_audio_init function in sound/usb/caiaq/audio.c and (2) the snd_usb_caiaq_midi_init function in sound/usb/caiaq/midi.c.	likely length-related and therefore likely prevented by checking packet length, which is built in
CVE-2013-1285	The USB kernel-mode drivers in Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2, R2, and R2 SP1, Windows 7 Gold and SP1, Windows 8, and Windows Server 2012 do not properly handle objects in memory, which allows physically proximate attackers to execute arbitrary code by connecting a crafted USB device, aka "Windows USB Descriptor Vulnerability," a different vulnerability than CVE-2013-1286 and CVE-2013-1287.	likely length-related and therefore likely prevented by checking packet length, which is built in

Cont'd

Table A.5 – *Cont'd*

CVE ID	Summary	Justification
CVE-2013-1286	The USB kernel-mode drivers in Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2, R2, and R2 SP1, Windows 7 Gold and SP1, Windows 8, and Windows Server 2012 do not properly handle objects in memory, which allows physically proximate attackers to execute arbitrary code by connecting a crafted USB device, aka "Windows USB Descriptor Vulnerability," a different vulnerability than CVE-2013-1285 and CVE-2013-1287.	likely length-related and therefore likely prevented by checking packet length, which is built in
CVE-2013-1287	The USB kernel-mode drivers in Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2, R2, and R2 SP1, Windows 7 Gold and SP1, Windows 8, and Windows Server 2012 do not properly handle objects in memory, which allows physically proximate attackers to execute arbitrary code by connecting a crafted USB device, aka "Windows USB Descriptor Vulnerability," a different vulnerability than CVE-2013-1285 and CVE-2013-1286.	likely length-related and therefore likely prevented by checking packet length, which is built in

Cont'd

Table A.5 – *Cont'd*

CVE ID	Summary	Justification
CVE-2013-3200	The USB drivers in the kernel-mode drivers in Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows Server 2012, and Windows RT allow physically proximate attackers to execute arbitrary code by connecting a crafted USB device, aka "Windows USB Descriptor Vulnerability."	likely length-related and therefore likely prevented by checking packet length, which is built in
CVE-2014-8884	Stack-based buffer overflow in the ttusbdecfe_dvbs_diseqc_send_master_cmd function in drivers/media/usb/ttusbdec/ttusbdecfe.c in the Linux kernel before 3.17.4 allows local users to cause a denial of service (system crash) or possibly gain privileges via a large message length in an ioctl call.	likely length-related and therefore likely prevented by checking packet length, which is built in
CVE-2005-2388	Buffer overflow in a certain USB driver, as used on Microsoft Windows, allows attackers to execute arbitrary code.	likely length-related and therefore likely prevented by design

Cont'd

Table A.5 – *Cont'd*

CVE ID	Summary	Justification
CVE-2006-1368	Buffer overflow in the USB Gadget RNDIS implementation in the Linux kernel before 2.6.16 allows remote attackers to cause a denial of service (kmalloc'd memory corruption) via a remote NDIS response to OID_GEN_SUPPORTED_LIST, which causes memory to be allocated for the reply data but not the reply structure	likely length-related and therefore likely prevented by checking packet length, which is built in
CVE-2006-6881	Buffer overflow in the Get_Wep function in cofvnet.c for ATMEL Linux PCI PCMCIA USB Drivers drivers 3.4.1.1 corruption allows attackers to execute arbitrary code via a long name argument.	prevented by checking packet length, which is built in

Appendix B

Code

This appendix describes the code I produced for this dissertation. For those keeping track at home, it amounts to 6410 lines of code (including comments and whitespace), of which 1061 were automagically generated.

B.1 Injection

As described in Chapter 3, I wrote the software stack that drives the Facedancer USB emulation board, which has since been used as the basis for industry-leading security-analysis tools such as umap [21].

I wrote both the library and, to demonstrate its use, a number of sample applications. The files comprising the library are enumerated in Table B.1 and the applications are enumerated in Table B.2. As the filenames imply, all code is written in Python.

All the code is available in the public GoodFET repository on github: `https://github.com/travisgoodspeed/goodfet`, under the "client" subdirectory.

B.2 Inspection

The instrumentation described in Chapter 4 touched a number of files in the FreeBSD kernel. Table B.3 shows the extent of the modifications made to each file.

I also wrote a number of scripts to analyze the output of the instrumentation. Those scripts are described in Table B.4 and are variously Bash, Python, awk, gnuplot, and D (the language used by DTRACE to manipulate the instrumentation probes).

filename	LOC	description
USBClass.py	25	Defines base class (in the object-oriented sense) for defining classes (in the USB sense) of devices.
USBConfiguration.py	46	Defines base class for describing USB device configurations.
USBDevice.py	360	Defines base class for describing USB devices (as evidence by its size, the majority of logic is in this class).
USBEndpoint.py	76	Defines base class for describing USB communication endpoints, analogous to sockets in network programming.
USBInterface.py	98	Defines base class for describing USB interfaces.
USB.py	53	Defines constants used throughout the rest of the library.
USBVendor.py	24	Defines base class for describing USB vendors.
total	682	

Table B.1: Files comprising the library I wrote to enable emulating various USB devices using the Facedancer.

filename	LOC	description
USBFtdi.py	201	Emulates a USB FTDI (serial) device.
facedancer-ftdi.py	22	Driver program for USBFtdi.py.
USBKeyboard.py	96	Emulates a USB keyboard.
facedancer-keyboard.py	21	Driver program for USBKeyboard.py.
USBMassStorage.py	356	Emulates a USB mass-storage device (e.g., thumbdrive).
facedancer-umass.py	41	Driver program for USBMassStorage.py.
total	737	

Table B.2: Applications I wrote to emulate various devices. All build on the libraries listed in Table B.1.

filename	LOC
usb_busdma.c	269
usb_compat_linux.c	9
usb_dev.c	18
usb_device.c	509
usb_dynamic.c	6
usb_hub.c	121
usb_lookup.c	22
usb_msctest.c	55
usb_parse.c	56
usb_pf.c	28
usb_process.c	30
usb_request.c	304
usb_transfer.c	614
usb_util.c	43
total	2083

Table B.3: Extent of modifications made to apply instrumentation framework described in Chapter 4.

filename	LOC	description
bb-count.d	12	D script to count number of basic-blocks executed.
bb-lint.py	163	Python script to verify the placement of instrumentation probes in the files listed in Table B.3.
bb-perc.sh	97	Shell script that calculates the percentage of basic blocks exercised, by file, from a given basic-block trace.
bb-trace.d	50	D script to trace functions called and basic blocks executed while a device is plugged into a particular USB port.
cam-trace.d	87	D script to trace interactions with FreeBSD's storage subsystem while a USB mass storage device is plugged into a particular USB port.
cloc-code.sh	18	Counts lines of code, used as input to other scripts.
extract-bb.sh	31	Shell script that extracts and format basic-block information output by bb-trace.d.
fbt-count.d	12	D script to count functions called in FreeBSD's USB stack; replaced unreliable built-in function boundary testing probes.
fbt-trace.d	8	D script to trace functions called.
group-bb-trace.py	54	Python script that groups the basic-block traces according to module.
indent-bb-trace.py	76	Python script that indents a basic-block trace for easier reading (example output shown in Figure 4.3).
make-graph.py	52	Python script that uses the result of group-bb-trace.py to generate a graphviz plot of module (file) interactions (example output shown in Figure 4.5).
mux-trace.d	56	D script to trace execution paths specifically surrounding the invocation of callback functions.
normalize.awk	9	Awk script to cleanly format the output of other scripts, used as input to still more scripts.
plot.p	12	Gnuplot script to graph per-function basic-block activity as generated by other scripts.
total	737	

Table B.4: Scripts used to interact with the instrumentation framework described in Chapter 4 and to process its output.

filename	LOC	description
makePolicy.hs	10	Driver program to create a loadable policy module from a plain-text policy description.
msgDriver.hs	10	Driver program to generate the header and source files that comprise the USB firewall.
Protocol.hs	238	Library that performs the automagic generation.
USBMessages.hs	279	Contains the USB message definitions.
total	537	

Table B.5: Hand-written source files for programs that generate the USB firewall.

filename	LOC	description
usb_messages.h	526	Contains data structure definitions, accessor macros, and function prototypes.
usb_messages.c	535	Contains validator functions and pretty-printing functions.
total	1061	

Table B.6: The generated USB firewall source files.

B.3 Generation

The Haskell code I wrote to generated the USB firewall is described in Table B.5. The resulting, generated code is described in Table B.6. The code I wrote to integrate the generated code with FreeBSD is described in Table B.7.

filename	LOC	description
opt_usb.h	6	Stub header file necessary to include other kernel-related header files.
usb_conntrack.c	216	Contains the connection-tracking code.
usb_conntrack.h	16	Header file for previous.
usb_fw.c	94	Contains the code that implements the basic USB firewall.
usb_fw_fbsd.c	147	FreeBSD-specific shim that acts as a translation layer between the structures used by FreeBSD to represent USB transfers and the OS-agnostic USB firewall.
usb_fw.h	19	Header file for the OS-agnostic USB firewall code.
util.c	64	Contains utility functions such as `bytes_as_hex`, which gives the contents of an arbitrary chunk of memory as hex.
util.h	12	Header file for previous.
total	573	

Table B.7: Hand-written source files that facilitate integration of the USB firewall with FreeBSD.

Bibliography

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. "An Overview of the Saturn Project". In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 2007.

[2] Stefan Axelsson. *Intrusion Detection Systems: A Survey and Taxonomy*. Tech. rep. Department of Computer Engineering, Chalmers University of Technology, 2000.

[3] Pablo Neira Ayuso. "Netfilter's connection tracking system". In: *USENIX ;login:* 31.3 (June 2006).

[4] Andrea Barisani. "Forging the USB armory". In: *Proceedings of the 31st Chaos Communications Conference (31c3)*. 2014.

[5] *Beagle USB 12 Protocol Analyzer*. URL: `http://www.totalphase.com/products/beagle-usb12/`.

[6] Nikita Borisov, David J. Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. "A Generic Application-Level Protocol Analyzer and its Language". In: *Proceedings of the 14th Annual Network & Distributed System Security Symposium*. 2007.

[7] Edwin C. Brady. "IDRIS: Systems Programming Meets Full Dependent Types". In: *Proceedings of the 5th ACM Workshop on Programming Languages meets Program Verification*. 2011.

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

[9] Sergey Bratus, Meredith L. Patterson, and Daniel P. Hirsch. "From "Shotgun Parsers" to Better Software Stacks". In: *SchmooCon IX*. 2013.

[10] Henning Brauer. "OpenBSD's pf: Design, Implementation, Future". In: *Proceedings of ruBSD 2013*. 2013.

[11] Suhabe Bugrara and Dawson Engler. "Redundant State Detection for Dynamic Symbolic Execution". In: *Proceedings of the 2013 USENIX Annual Technical Conference*. 2013.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 2008.

[13] US-CERT/NIST. *Vulnerability Summary for CVE-2013-1285*. Feb. 2013. URL: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1285`.

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An empirical study of operating systems errors". In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 2001.

[15] Gerald Combs. *wireshark*. 2006. URL: `www.wireshark.org`.

[16] Tool Interface Standards Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) specification version 1.2*. May 1995.

[17] Kees Cook. *USB AVR fun*. 2012. URL: `http://outflux.net/blog/archives/2012/05/16/usb-avr-fun/`.

[18] Microsoft Corporation. *User Mode Driver Framework*.

[19] Microsoft Corporation. *Vulnerabilities in Kernel-Mode Drivers Could Allow Elevation Of Privilege*. Mar. 2013. URL: `https://technet.microsoft.com/library/security/ms13-027`.

[20] Andy Davis. *Lessons learned from 50 bugs: Common USB driver vulnerabilities*. Tech. rep. NCC Group, 2013.

[21] Andy Davis. *umap — the USB host security assessment tool*. 2013. URL: `https://github.com/nccgroup/umap`.

[22] Andy Davis. "USB: Undermining Security Barriers". In: *Proceedings of the BlackHat Technical Security Conference*. 2011.

[23] *Diagrams Haskell library*. 2014. URL: `http://projects.haskell.org/diagrams/`.

[24] Kathleen Fisher and Robert Gruber. "PADS: a domain-specific language for processing ad hoc data". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005.

[25] USB Implementers Forum. *Universal Serial Bus Specification Version 2.0*. Apr. 2000.

[26] Travis Goodspeed. "iPod Antiforensics". In: *PoC or GTFO* 12.0x00 (Aug. 2013).

[27] Travis Goodspeed and Sergey Bratus. “Facedancer USB: Exploiting the Magic School Bus”. In: *Proceedings of the REcon 2012 Conference*. 2012.

[28] Travis Goodspeed, Sergey Bratus, Ricky Melgares, Ryan Speers, and Sean W. Smith. “Apido: Tools for Exploring the Wireless Attack Surface in Smart Meters”. In: *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*. 2012.

[29] Ilfak Guilfanov. *IDA*. 2015. URL: `https://www.hex-rays.com/products/ida/index.shtml`.

[30] Mark Handley, Vern Paxson, and Christian Kreibich. “Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics”. In: *Proceedings of the 10th USENIX Security Symposium*. 2001.

[31] Maxim Integrated. *MAX3421E USB Peripheral/Host Controller with SPI Interface*. URL: `https://www.maximintegrated.com/en/products/interface/controllers-expanders/MAX3421E.html`.

[32] Alex Ionescu. “Hooking Nirvana: Stealthy Instrumentation Techniques for Windows 10”. In: (2015).

[33] Van Jacobson, Craig Leres, and Steven McCanne. *tcpdump*. 1987. URL: `www.tcdump.org`.

[34] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Tech. rep. AT&T Bell Laboratories, 1975.

[35] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 2009.

[36] F-Secure Labs. *F-Secure: Stuxnet Questions and Answers*. Oct. 2010. URL: `https://www.f-secure.com/weblog/archives/00002040.html`.

[37] Daan Leijen and Erik Meijer. *Parsec: Direct style monadic parser combinators for the real world*. Tech. rep. Department of Computer Science, Utrecht University, 2001.

[38] Jacob Maskiewicz, Benjamin Ellis, James Mouradian, and Hovav Shacham. “Mouse Trap: Exploiting Firmware Updates in USB Peripherals”. In: *Proceedings of the 8th Annual Workshop on Offensive Technologies (WOOT 2014)*. 2014.

[39] Peter J. McCann and Satish Chandra. “Packet Types: Abstract Specification of Network Protocol Messages”. In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 2000.

[40] Haroon Meer. "Memory Corruption Attacks: The (almost) Complete History". In: *Proceedings of the BlackHat Technical Security Conference*. 2010.

[41] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Science* 17 (1978).

[42] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2012.

[43] *MQP Electronics Packet-Master USB500 AG+*. URL: `http://www.mqp.com/usb500.htm`.

[44] Randall Munroe. *Exploits of a Mom*. `http://xkcd.com/327/`.

[45] Aleph One. "Smashing the Stack For Fun and Profit". In: *Phrack* 7.49 (Nov. 1996).

[46] *OpenSSL Security Advisory: TLS heartbeat read overrun (CVE-2014-0160)*. 2014. URL: `https://www.openssl.org/news/secadv/20140407.txt`.

[47] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. "binpac: a yacc for Writing Application Protocol Parsers". In: *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. 2006.

[48] Meredith Patterson. "Guns and Butter: Towards Formal Axioms of Input Validation". In: *Proceedings of the BlackHat Technical Security Conference*. 2005.

[49] Vern Paxson. "Bro: A System for Detecting Network Intruders in Real-Time". In: *Proceedings of the 7th USENIX Security Symposium*. 1998.

[50] Jon Postel, ed. *Internet Protocol DARPA Internet Program Protocol Specification*. RFC 791. Sept. 1981. URL: `http://www.ietf.org/rfc/rfc791`.

[51] The FreeBSD Project. *FreeBSD*. URL: `https://www.freebsd.org`.

[52] Thomas H. Ptacek and Timothy N. Newsham. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*. Tech. rep. Secure Networks, Inc., 1998.

[53] redpantz. "The Art of Exploitation: MS IIS 7.5 Remote Heap Overflow". In: *Phrack* 12.68 (Apr. 2012).

[54] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (Mar. 1953).

[55] Umesh Shankar and Vern Paxson. "Active Mapping: Resisting NIDS Evasion Without Altering Traffic". In: *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. 2003.

[56] A. Prasad Sistla, V. N. Venkatakrishnan, Michelle Zhou, and Hilary Branske. "CMV: Automatic Verification of Complete Mediation for Java Virtual Machines". In: *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. 2008.

[57] *Solaris Dynamic Tracing Guide*. Santa Clara, CA: Sun Microsystems, 2008.

[58] National Institute of Standard and Technology (NIST). *CWE - Common Weakness Enumeration*. URL: `https://nvd.nist.gov/cwe.cfm`.

[59] National Institute of Standard and Technology (NIST). *National Vulnerability Database (NVD)*. URL: `https://web.nvd.nist.gov/view/vuln/search`.

[60] Vasily Tarasov, Erez Zadok, and Spencer Shepler. "Filebench: A Flexible Framework for File System Benchmarking". In: *USENIX ;login:* 41.1 (Spring 2016).

[61] *Teensy USB Development Board*. URL: `https://www.pjrc.com/teensy/`.

[62] Jacob Torrey and Mark Bridgman. "Verification State-space Reduction through Restricted Parsing Environments". In: *Proceedings of the 2015 Workshop on Language-Theoretic Security*. 2015.

[63] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits". In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. 2004.

[64] Yan Wang. "A Domain-Specific Language for Protocol Stack Implementation in Embedded Systems". PhD thesis. Örebro University, 2011.

[65] Yan Wang and Verónica Gaspes. "A Domain-Specific Language Approach to Protocol Stack Implementation". In: *Practical Aspects of Declarative Languages*. 2010.

[66] Yan Wang and Verónica Gaspes. "An Embedded Language for Programming Protocol Stacks in Embedded Systems". In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2011)*. 2011.

[67] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. "Using CQUAL for Static Analysis of Authorization Hook Placement". In: *Proceedings of the 11th USENIX Security Symposium*. 2002.