Protecting Against Malicious Bits On the Wire

Automatically Generating a USB Protocol Parser for a Production Kernel

Peter C. Johnson
Middlebury College
Middlebury, Vermont

Sergey Bratus
Dartmouth College
Hanover, New Hampshire

Sean W. Smith
Dartmouth College
Hanover, New Hampshire

ABSTRACT

Recent efforts to harden hosts against malicious USB devices have focused on the higher layers of the protocol. We present a domain-specific language (DSL) to create a *bit-level* model of the USB protocol, from which we automatically generate software components that exhaustively validate the bit-level syntax of protocol messages. We use these generated components to create a stateful, connection-tracking firewall for USB. We integrate this firewall with the FreeBSD kernel and demonstrate that it achieves complete mediation of USB traffic, thus protecting the rest of the kernel, including higher-level policy mechanisms such as USBFilter, from low-level attacks via maliciously crafted packets.

In addition to in-kernel data structures and packet validation routines, our system generates a user-level policy engine that allows for flexible and expressive firewall behavior beyond mere message syntax validation, as well as functions for pretty-printing packets (which can be used in both the kernel and in protocol analysis software). We use a Haskell back-end to generate C code that we integrate with the FreeBSD kernel, thus making our entire system amenable to formal verification.

ACM Reference Format:
Peter C. Johnson, Sergey Bratus, and Sean W. Smith. 2017. Protecting Against Malicious Bits On the Wire: Automatically Generating a USB Protocol Parser for a Production Kernel. In ,. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3134600.3134630

1 INTRODUCTION

We are pleased to see the USB attack surface receiving recent attention (e.g., USBFilter [38], Cinch [8]), but we feel a vital piece of the puzzle is missing: namely, verifiable, bit-level parsing of protocol messages. Frameworks to enforce policy on messages passing over the Universal Serial Bus can protect against malicious or misbehaving devices only when the contents of those messages can be accurately interpreted. Otherwise, we have a situation where the guard is checking IDs, but not making sure the holder actually belongs to the ID card!

The aforementioned systems provide means to control access to services provided over USB—but that control is exercised at the protocol level: they limit requests that are conveyed over USB *via syntactically correct packets*. They model functionality at the device and session levels, and thus fill an important gap in controlling access and functionality: they create concise and actionable descriptions of wanted and unwanted protocol sessions, and enforcement mechanisms to turn these descriptions into policy. They do not, however, ensure the individual messages are well-formed.

It is at the bit level that USB presents another attack surface, explored to date primarily by industry: attacks on kernel code via crafted, malformed USB messages [7, 12, 15]. This attack surface also needs a firewall, and this is the firewall we provide. It is complementary to a system like USBFilter and, as we discuss below, should be combined with it for stronger security guarantees.

Our approach comes from the view that protocol data is a language to be modeled. From the model, we develop *recognizer* code to accept or reject USB packets and data structures as instances of this language. This approach to firewall code makes it amenable to formal verification; it is no longer a loose collection of "sanitization" heuristics or "sanity checks" on input data, nor an informal interpretation of the protocol standard; it is an automaton with a precise specification that can be verified.

Our code is integrated in a production kernel, serving as evidence that our approach is practical.

1.1 Why extant approaches aren't enough

Systems like USBFilter may appear to have solved the problem of protecting USB stacks from malicious payloads: the firewall would block malicious messages. Yet this is not the case. The kernel's USB stack is still unprotected where it is most vulnerable: while parsing USB's most complex data—e.g., enumeration descriptors. Such a parser is USB's biggest attack surface, and it lies below the level of USBFilter; in fact, USBFilter depends on its correctness to characterize devices to which policy is applied. Once USB enumeration code is exploited, it is too late to apply a policy.

Such vulnerabilities continue to be a risk even for userland or emulated stacks, which may shift the locus of the vulnerability and mitigate its effects, but do not remove its root cause: weak parsing or, more precisely, memory-unsafe kernel code exposed to hostile data that acts on unchecked assumptions and corrupts memory.

These vulnerabilities historically abound. Consider, for example, a bug in which a field of the USB hub descriptor, expected to contain an integer no larger than 127, caused a buffer overflow when set to 0xFF [4]. So did crafted values of *wMaxPacketSize* [3], unchecked disagreements between *bLength* and *wTotalLength* [5] and many similar bugs across different operating systems. Our analysis of the CVEs shows that these memory corruption bugs in early-stage parsing of USB messages continue to occur. Thus descriptor parsing remains an important, unprotected part of the attack surface.

ACSAC 2017, 2017 Annual Computer Security Applications Conference

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134630

While systems like Cinch change the execution model of USB stacks to a safer virtualized one, they don't obviate the need for safe bare-metal OS code. Our approach, which produces just such code, has additional benefits even for policy application such as USBFilter that occurs after enumeration. Namely, modeling establishes all, not just a few, allowed USB payloads. For example, USBFilter's example module for write-protecting a USB drive blocks writes that use SCSI's `WRITE(10)` command, but could be bypassed by, e.g., `WRITE(6)` or `WRITE(12)`. Our approach would establish and enforce a full subset of allowed commands as a USB protocol (sub)language in the policy design stage—and thus prevent such oversights that are inevitable when dealing with a complex protocol.

1.2 Contributions

a) We developed a data model for the USB protocol and expressed it in a domain-specific language, implemented in Haskell.
b) From this model, we generated C code to validate USB packets.
c) We integrated this code in the FreeBSD kernel and instrumented the kernel with a set of DTrace probes for the USB subsystem.
d) We demonstrated experimentally that our generated code performant and suitable for inclusion in production kernels.
e) We surveyed all USB-related vulnerability disclosures and classified them according to whether our validation mechanism would mitigate their effects.

In short, we present a firewall for USB packets, implemented as a patch for the FreeBSD kernel. Our method is complementary to USBFilter, and should be used in concert with it, to protect the kernel from malformed USB packets that USBFilter assumes.

2 USB: PROTOCOL AND VULNERABILITIES

The USB protocol consists of packets on the wire that are parsed (primarily) by the OS kernel. Thus, different as its wiring and controller chips may be from the TCP/IP networking protocols, USB implementations have the same low-level nemesis: parser bugs and memory corruption triggered by crafted packets.

For TCP/IP, this threat loomed large in 1990s—see, e.g., *Teardrop* and *Land* single-packet-crash attacks, as well as other "pings of death" [29]—and was mitigated by hardening kernel stack parsers and dropping packets that don't conform to a minimal subset of the protocol. Even so, crafted IP packets are still capable of occasional nasty surprises (e.g., *Darwin Nuke* [20]).

For USB, the interaction of crafted packets with parsers is still a primary threat, as fuzz-testing has shown. For example, Davis discusses over 50 bugs triggered by fuzzed USB messages [7], injected with a rough equivalent of "raw sockets" for USB [12, 19]. Doubtless, further modeling of the USB protocol and more advanced exploration tools such as *Umap2* [28], will turn up more bugs.

Thus it is important to review where in the protocol these bugs occur, before we present our antidote.

2.1 The Protocol

When a USB device is plugged in, the host must determine whether it is, e.g., a keyboard, a mouse, a printer. This initial conversation between host and device, called *enumeration*, happens for every device. In addition to determining characteristics of the device such as polling frequency and preferred data transfer size, the host also decides which kernel driver to associate with the device when enumeration is complete. The communication channel that carries these messages is separate from application 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 (*endpoints*) to and from the device.

```
Host to device (request)   Device to host (response)
80 06 00 01 00 00 12 00    12 01 00 02 00 00 00 40 1E
                           04 02 04 00 01 01 02 03 01
```

Figure 1: A USB request/response pair.

This gives the device significant leverage over the kernel: it gets to choose precisely which driver will 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 immediately gives a communication channel straight to the kernel. Even following enumeration, most application-level USB drivers still run in kernel mode as well (though this is changing: see Microsoft's User Mode Driver Framework [14]). Thus—unlike TCP/IP!—USB exposes the kernel parser attack surface not only of the USB subsystem proper, but of many other drivers and subsystems as well.

Most of the messages that comprise enumeration are *descriptors* that contain various parameters of the device in question. Figure 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 request identifies the requested descriptor (here, the "device" descriptor) and the seventh byte indicates the amount of data the host would like to receive back (0x12 = 18 bytes).

In the response, the first byte indicates its length, which 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 Heartbleed [6] vulnerability exemplifies buffer underflows and overflowed buffer instances are legion [30, 34].)

In this work, we focus on the enumeration phase, as the phase that contains the most complex parsers and has been found to harbor a surprisingly large number of bugs.

2.2 USB as a Gateway to the Kernel

Like the TCP/IP networking protocols, USB is *layered* in that it allows data from one protocol to be encapsulated inside another. This is how USB supports 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 the past (e.g., SCSI, audio, keyboards), this often provides a direct codepath to parts of the kernel outside the USB stack itself.

In 2012, we explored the reachability of kernel logic from the USB interface with a focus on the storage subsystem, down to the

granularity of basic blocks [12]. Our work showed that a USB device could access essentially the entire FreeBSD storage subsystem, which is notable because so many other aspects of the system depend on disks. Furthermore, the storage subsystem is just one of the many subsystems in the kernel that are accessed by USB devices, such as 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. Parsing is a crucial boundary to ensuring the security of running systems.

2.3 Vulnerabilities

In 2013, Andy Davis released *Umap* [17], a test suite to explore the behavior of USB stacks in the face of unexpected input received during enumeration. Umap, and its successor Umap2 [28], play a key role in evaluating our system being, to the best of our knowledge, the most complete suite of USB malformation patterns aimed at triggering different classes of USB bugs.

Using special-purpose Facedancer [19] hardware, he emulated a variety of USB devices getting plugged into a target host, and controlled every aspect of the data sent from these devices to the host during enumeration. 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, from which he deduced an ontology of bugs [7], which we summarize in Table 1.

2.4 National Vulnerability Database

The National Vulnerability Database captures a kind of "ground truth" of existing vulnerabilities. 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, though it is searchable.

Between January 2005 and December 2015, exactly 100 of the vulnerabilities reported to the NVD contained the string "usb". We surveyed all 100 of these vulnerabilities, placed each in one of the five categories identified by Davis, and use these in our evaluation of our firewall's effectiveness. Not surprisingly, the bugs mitigated by proper parsing form a significant subset: nearly half of all USB-related vulnerabilities!

3 PROTOCOL MODELING

To a kernel developer, a binary protocol implementation starts with a C header file defining the protocol's data elements as C structs. These definitions are used to declare variables, but their most prominent use comes from accessing individual fields of protocol messages by casting a pointer to a raw byte buffer to the struct pointer type and then dereferencing it to extract a field value. Thus the C struct definitions are already, in a sense, active parser code, as well as a de-facto translation of a protocol specification. What are they (and the programmers) missing that manifests itself as vulnerabilities and exploitable bugs plaguing C code?

First, these structs describe the components of a protocol's message, but not their *relationships*. Left unspecified, these relationships are left to be checked unsystematically, typically just before a piece of data is used, or never at all. Moreover, the majority of relevant code assumes that the relationships have been checked; as Morris noted in 1973, the programmer "*could begin each operation with a well-formedness check, but in many cases the cost would exceed that of the useful processing.*" [21]

This lack of method as to where and how to perform the checks is a major source of code weaknesses. For example, nested length fields (length fields on objects that contain other objects, also with length fields that must agree with the containing object's), have been a major cause of USB bugs (see Davis' analysis [7] and ours in Section 5.4). Together, these element definitions and their relationships form the syntax of the protocol's messages. Thus a crucial part of this syntax fails to get specified consistently.

Second, the structs describe what amount to *finite languages* in the Chomsky syntactic hierarchy, whereas it's natural for protocol designers to use at least *regular language* syntax (e.g., constructs like "one or more objects of this type"), let alone context-sensitive object nesting constructs. Failure to check the syntax of inputs with the automaton appropriate to the actual input language is a major cause of input-handling bugs [35].

In fact, it is often not clear to developers what kind of a computational task they should implement when checking inputs and what algorithms are appropriate for the task. Lack of clarity leads to ad-hoc and inadequate algorithms. Yet a solid formal theory of syntax checking exists and, moreover, warns of algorithmic pitfalls, with its signature results establishing a hierarchy of formal languages and their recognizer automata. It should be used, and we base our approach to input validation on it.

Thus, in order to produce a better kind of input-handling code, we need to start with a different kind of data definitions, which capture both the identities of protocol fields and their relationships, allowing a systematic way of constructing the code to recognize and validate the messages as instances of an input language.

3.1 Choice of Tools

We chose Haskell in which to embed our domain-specific language (DSL) to model USB data, for several reasons.

Ease of DSL development. Haskell's ability to manipulate its own syntax gives us powerful and flexible means to define a data DSL.

Strongly typed. Haskell's type system provides a powerful means of modeling USB data objects, which maps well to describing USB messages as a input language.

Potential for formal verification. Haskell code maps well to that of proof assistants such as Isabelle/HOL and Coq.

In this paper, we model the data structures for USB enumeration; we use the `GET_DESCRIPTOR` request message sent from host to device as a running example, and show the path from its definition to the generated C data structs that represent it, and the code that handles it and integrates into the FreeBSD kernel.

3.2 Protocol Model

Message. We define a Protocol to be a set of *Messages*, each of which consists of a name, some Fields, and a data stage of variable length. (Figure 2 shows its definition using our domain-specific language.) Though simple, this definition describes all messages

Class	Description
Unspecified denial of service	The driver or host machine usually crashes, but not in a way that is exploitable by an attacker.
Buffer overflows	Bounds are not adequately checked prior to memory operations.
Length-related bugs	Arithmetic performed on values provided by the device can lead to unintentional memory allocations.
Format string bugs	User-controlled input is used as the format string in calls to the `printf` family of functions.
Logic errors	The operating system incorrectly handles a given input.

Table 1: Classification of USB-related vulnerabilities, due to Andy Davis.

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

Figure 2: Specification of a Message within a protocol.

```
data Field = Field FieldName FieldSize FieldValue

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

Figure 3: Specification of a Field within a Message.

exchanged during enumeration, dubbed "control messages". Following enumeration, most communication is application-specific, and supporting such protocols using our 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 name is a character string used for identification purposes. It is followed by a list of Fields (described below) that make up the Message. Fields are assumed to be ordered and contiguous within the Message; should the protocol specify empty space or padding, one would specify an explicit Field reflecting those characteristics.

We express these and further relationships between message elements in a grammar that is also the Haskell definition of the types within the protocol DSL. Message and field names are 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 [26].

Field. Each *Field* consists of its name, size, and whether its contents are literal or variable. (Its definition is shown in Figure 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.

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 0x4. Likewise, USB requires that `GET_DESCRIPTOR` request messages have a RequestType field of 0x80 and a Request value of 6. These are specified as Literal fields, along with the value they require.

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

Figure 4: Specification of the length of the data stage of a Message.

Alternatively, some fields are not proscribed and must be available to higher-level code that, e.g., changes state within the kernel upon receipt. 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 are specified as Variable fields so that the appropriate code can be generated.

DataLen. Message fields may be followed a data stage, the definition of which is shown in Figure 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 (e.g., the `GET_DESCRIPTOR` request) specify the length in one of the Fields, in which case the "Ref" constructor is used. If, for example, the length is specified in the `wLength` field, the DataLen constructor would appear as `Ref "wLength"`.

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

3.3 Example: `GET_DESCRIPTOR` Request Message

We use the `GET_DESCRIPTOR` request message as an example; its specification is shown in Figure 5. The first parameter specifies the name (to be used throughout 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, 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.

```
getDescriptorRequest :: Message
getDescriptorRequest = Message "GET_DESC 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: Specification of USB's `GET_DESCRIPTOR` request message, written in the domain-specific language.

```
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;
};
```

Figure 6: C structure generated from the Get Descriptor request message specified in 5.

3.4 Other DSL Features

We leave other aspects of our domain-specific language for Appendix A, as they are not directly germaine to the correctness of our firewall (though they do aid ease of use).

4 KERNEL CODE GENERATION

A great deal of code—in fact, we argue much of the protocol-specific code in the kernel—can be automatically generated from the model described in the previous section. As a result, much of the ad-hoc code that comprises most protocol stack implementations can be replaced with generated code. Although generated code is not inherently superior, it does provide some significant potential benefits.

Why Code Generation? First, secure coding practices (e.g., bounds checking or agreement between length fields of objects contained in other objects) can be encoded in the generation logic, meaning that *every* field in *every* message of *every* stack that is generated using the framework will be bounds-checked. Bugs arising from, e.g., accidentally neglecting to check the bounds of a particular field, thus allowing a buffer overflow, simply cannot happen. Programmers need not write these checks manually, at the risk of forgetting some—they now only need to specify that a relationship exists, and a check will be generated for it.

Second, newly-developed programming techniques can be encoded in the generation logic, all stacks can be re-generated, and *every* stack will immediately benefit. Maintainers of individual protocol stacks no longer have to understand the new technique, pore over their code to discover where to apply it, and patch it manually.

Third, given that the protocol model is amenable to formal verification, any new protocol implementaions generated from it will be ready for verification as well, such as described in 6.3. Moreover, additional annotation can be added to the model as needed and generated along with code, avoiding the much more laborious process of adding it to the existing code.

Generated Components. With these benefits in mind, we can generate the following different code components from our message definitions, described in Section 3.2.

- data structure definition
- parser/verifier
- field accessor functions
- pretty-printer functions
- user-defined policy engine

We proceed by describing the purpose of each, including examples derived from the `GET_DESCRIPTOR` request message defined in Figure 5. Where generated code is not directly germaine to the firewall, we defer details to the Appendices.

4.1 Generating the Data Structure

First, we need a data structure to represent each message. This structure can (and likely should) be used both in the kernel proper and the parsing component. Additionally, it could be used in devices that inject protocol data such as the Facedancer and its associated software *umap*, as well as programs such as `tcpdump` that analyze protocol traces. Figure 6 shows the structure definition generated from the `GET_DESCRIPTOR response` message shown in Figure 6.

Defining fields for fixed-size types is straightforward: the `Uint8` and `Uint16` of the definition from Figure 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.

4.2 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 and aspects of the entire frame—most significantly, its length, so as to avoid vulnerabilities such as Heartbleed [6]. The generated parser function for the `GET_DESCRIPTOR` request message is shown in Figure 7.

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 instigating request, but the parser cannot know this without maintaining significant application-specific state. Such state is more the purview of a separate component that verifies the validity of a *sequence* of messages rather than each individual message in the sequence; this work is focused 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 the `conntrack` module.

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

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

Figure 8: Example user policies for the USB firewall.

4.3 Generating Accessor Functions

Although the data members of structures generated by the code described in Section 4.1 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. Elaboration, including examples, is given in Appendix A.1.

4.4 Generating the Pretty-Printer Functions

We also require the ability to present the details of a message in a user-friendly format. While perhaps not strictly needed by the kernel proper, this feature is vital to user-facing tools that inspect protocol traffic. (For example, a protocol-specific tool analogous to `tcpdump`.) See Appendix A.2 for further details and examples.

4.5 User-Defined Policies

A key feature in a firewall is the ability to specify a policy to augment the built-in syntax-checking rules. For instance, imagine a case where a particular USB device driver doesn't correctly handle a string descriptor with the 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.

We devised a simple language to describe policies (see Figure 8), 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 4.7.

These policies are, admittedly, not especially eloquent. In particular, they do not take into account the context in which a message is being sent. Notably, the USBFilter system with its models of contexts and transactions fills this gap. USBFilter provides a model and a language for expressing context-aware policy, that is, a means to specify behaviors and actions of USB devices. We anticipate that this model can be used to translate higher-level behavior descriptions to packet-level ones, which our firewall could enforce, in addition to its primary function, enforcing the USB specification or its subsets.

4.6 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, we defined instances of the `Message` type for the following messages:

- `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`

We 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 B.2).

4.7 Operating System Integration

The basic processing path of our firewall is shown in Figure 9. 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. At this point, the packet can be passed on to another system, such as USBFilter or Cinch, to perform semantic validation. For further details, see Appendix B.

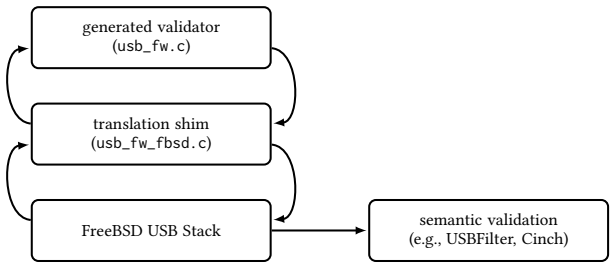

Figure 9: USB frame processing process.

USB id	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 2: Data sent by umap fuzz-testing.

5 EVALUATION

We set the following evaluation goals.

Stability We must show that the firewall does not crash in the face of both legitimate and malformed USB traffic.

Complete mediation We must show that all frames are examined for accordance with the policy.

Overhead We must show that the firewall performs its job without incurring a significant performance penalty.

Effectiveness We must show that the firewall correctly identifies and interdicts malformed USB frames.

5.1 Stability and Effectiveness.

Umap [17] was the primary tool we used to test the stability and effectiveness of our USB code; it is 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. Umap represented the state-of-the-art in testing the security of host-side USB implementations at the time of our testing. (Umap's successor, *Umap2*, has been recently released, and we plan to use it shortly.) Its test suite contains the largest set of known USB vulnerability triggers. All told, we ran nearly all its 500 different vulnerability triggers against our USB parser/firewall; a finer breakdown of the tests is shown in Table 2.

In our testing setup, the Facedancer hardware, 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. The target detects no device at all even when these connections are made. Only when software (e.g., umap) running on the host causes the Facedancer to emulate a particular device does the target actually see a device connect. Once that happens, the software on the host controls nearly all aspects of the emulated device's behavior (exceptions discussed below).

5.1.1 Identifying testable drivers. The umap software package supports many testing modes. We first ran its "identification" mode to determine which devices were supported by the FreeBSD target so we could focus on these in the remainder of our testing. Of the device classes testable by umap, six are supported in the FreeBSD target: *audio control*, *audio stream*, *human interface devices* (mice and keyboards), *printers*, *mass storage* (e.g., thumbdrives), and *hubs*.

With this identification running against our code (shown in Appendix **??**), we have also taken a first step to showing that the firewall is stable in the face of real USB traffic: our code did not ever crash while being probed by umap—*despite umap being a tool explicitly designed to cause such crashes!*

5.1.2 Fuzz-testing individual drivers. With these six device classes in hand, we proceeded to test each individually using umap's fuzz-testing feature, which causes the Facedancer to emulate a particular device and, as part of the USB enumeration phase, send frames that push 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` and another where it sends `0xFF`, to verify that the kernel safely handles extreme cases.

For each device class, umap supports a large number of such tests: we ran them all. Our USB firewall was configured with no user-policy rules; only the generated validation functions were invoked. Appendix **??** shows sample output from a single run.

We saw interesting behavior when using umap to test human interface devices (class 03:00:00). The firewall successfully recognizes and rejects umap's "Configuration_bDescriptorType_null" test, in which it 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 we elided them from the test suite.

Thus, rather than undermining our 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 was beyond the scope of our work, the root cause is likely non-trivial. The fact remains, however: our system discovered this bug.

Table 2 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.

This claim of stability might seem undermined by the HID behavior described at the beginning of this section. We contend it is

test	sent	processed	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 3: 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.

eminently reasonable: the firewall itself *did the correct thing* 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.

5.2 Complete Mediation

To demonstrate *complete mediation*, we must show that our code examines all the data that flows over the USB bus.

5.2.1 Umap Empirical Input/Output Test. To empirically test whether every single frame sent by umap is evaluated by the USB firewall, we configured umap to record every frame sent and we instrumented and configured the firewall to record every frame evaluated. Then we ran the entire fuzz-testing suite described in Section 5.1 and gathered the results shown in Table 3.

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 we had logged the raw bytes being sent by umap and received by the firewall, we were able to check the differences in actual data being sent and received. *Every single frame sent by umap was analyzed by the firewall.* 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. No exceptions.

5.2.2 Sniffing the Bus with Protocol Analyzer. To make these results conclusive, we used a Beagle USB 12 Protocol Analyzer [1] to capture all USB data sent by umap to the FreeBSD target. The Beagle sits between the Facedancer and the target, mirroring all USB data to a third host (see Figure 10). We 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

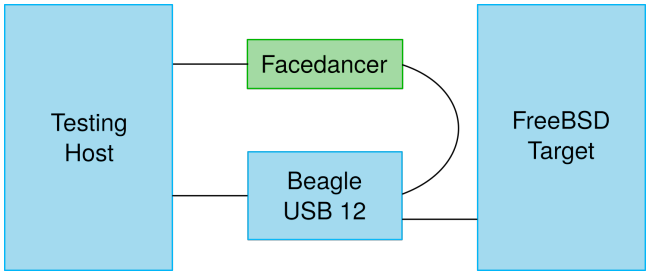

Figure 10: To capture USB data sent by the Facedancer, we connected the Beagle USB 12 Protocol Analyzer as a pass-through device between the Facedancer and the FreeBSD target; then we connected the Analysis port of the Beagle back to the testing host to capture packets.

Beagle and the set of USB packets mediated by the firewall. These discrepancies fell into two categories:

Repeated messages 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.

Multi-packet messages 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 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 our testing, no other discrepancies exist between USB data seen by the Beagle protocol analyzer and our USB firewall. Thus we have empirical evidence of complete mediation.

5.3 Performance

In addition to being stable and enforcing complete mediation, the firewall must not incur undue performance penalties. To evaluate the additional processing time induced by the presence of the firewall, we measured its impact on the *common case* performance, and *corner case* performance,

5.3.1 Corner Case Performance. We again used the umap fuzz-testing feature—this time around, timed—since umap's very purpose is to exercise corner cases. We modified umap to produce as little output as possible and we turned off all logging in the USB firewall. We then ran each test suite three times, rebooting between each test. Our test target was an Intel Core 2 Duo U7300 machine (2 cores, 1.3 GHz, 3 GB memory) and our test host was an Intel Core i7-4600U (4 cores, 2.1 GHz, 12 GB memory); the results for each set of test suite runs are shown in Table 4.

These numbers told an interesting 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

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

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

Table 5: Results of running FileBench's `singlestreamread` workload on a USB mass storage device, with the firewall enabled and disabled.

the disparities are so unevenly spread among device classes; the abysmal performance of the hub class is particularly worrisome.

We 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 we 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 the device. This mirrors the situation we discovered with human-interface devices (described in Section 5.1.2).

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. Further research into the "correct" abstraction to present seems worthwhile.

5.3.2 Common-Case Performance. The previous section describes 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 we also measured the impact of the firewall on legitimate traffic. We used the `singlestreamread` workload from the FileBench benchmark suite [36] to measure throughput of a USB mass storage device. We ran 5 experiments each with the firewall enabled and disabled. The results are shown in Table 5.

Oddly, FileBench reports that performance is *better* when the firewall is enabled compared to when it is *dis*abled. We hypothesize this may be due to code structure and caching effects; the numbers are so close, however, that there is essentially no difference. In any case, our generated USB firewall code does not incur an unreasonable performance penalty in the face of legitimate traffic.

5.4 Effectiveness against CVEs

To assess the promise of our approach to mitigate vulnerabilities in the wild, we turn to Common Vulnerability and Exposure (CVE) records as a proxy for ground truth. We surveyed all reports from January 2005 through December 2015 that contained the text "usb" and classified them according to their likely relation to errors in parsing.

Class	Quantity
Unrelated	45
Unclear	12
Mitigated by Policy	27
Mitigated by Design Pattern	3
Inherently Averted	14

Table 6: Classification of USB mentions in CVE incident reports by how they might be affected by our USB firewall. See Appendix D for enumeration of specific CVEs.

We reviewed each of these 100 vulnerabilities and categorized them based on whether and how the USB protection framework we 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. However, even the disclosures relatively devoid of details provide some hints.

Table 6 summarizes the five vulnerability categories we settled on and the vulnerabilities we assigned to each, as explained below. All told, the three categories addressed by our system in one way or another, make up nearly *half of all the USB-related CVE vulnerabilities*—even considering that many of the "unrelated" vulnerabilities only coincidentally mention USB! (The complete list of CVEs classified is given in Appendix D.)

Unrelated Almost half of the vulnerabilities turned up by the search only incidentally touched on USB. 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.

Unclear We were unable to categorize about 10% of the USB-related vulnerabilities in our 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. Moreover, Oracle's "disclosures" decline to specify any details, as exemplified by CVE-2011-2295.

Mitigated by Policy We 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.

Mitigated by Design Pattern Three vulnerabilities resulted from deviations from sound coding 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, CVE-2010-4074) and failure to clear transfer buffers before returning to userspace (CVE-2010-1083).

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, we assumed 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.

6 RELATED WORK

6.1 Attacks and protections for USB stacks

Increased attacker interest towards weaknesses in USB code can be traced to at least 2010–2011 (e.g., [13, 16]), such as the *PSGroove* jailbreak of the Play Station 3. These used reprogrammed USB devices such as AVR *Teensy* [2] to deliver the exploit payload; subsequent self-hosting platforms like *USB Armory* [9] significantly expanded delivery capabilities. Meanwhile, tools such as Facedancer [19] removed the need to reflash and reconnect a delivery device, allowing faster iteration through attack scenarios and payloads, and further methods to scale up their use have been proposed [39]. Attacks on devices were also considered (e.g., [23]).

Recently, GoodUSB [37], USBFilter [38], and Cinch [8] proposed different approaches to protect devices from attacks. GoodUSB uses a user-in-the-loop approach that constrained enumerable functions based on how the device was identified. USBFilter, inspired by NetFiler, introduced a kernel architecture aimed at controlling unwanted access to services by USB peripherals. Cinch, by contrast, leverages virtualization. As we discussed in the introduction, our approach is complementary to USBFilter's; USBFilter represents a significant step in USB policy in production kernel code, but should be used in conjunction with a packet-level recognizer protecting it and the rest of the kernel from attacks by crafted packets.

6.2 DSLs and auto-generated message parsers

Domain specific languages (DSL) have been used for *describing* the contents of network messages since the 1990s, including in BSD Packet Filter (BPF) [25] (used by, e.g., *tcpdump*), the Bro IDS [32], and others. These languages, however, were intended for generating auxiliary packet analyzer code that extracted certain features from packets, not for generation of comprehensive packet parsers for kernel stacks. Indeed, even with these DSLs around, packet analyzer tools such as tcpdump contained ad-hoc parsers with DoS and remote code execution vulnerabilities such as CVE-1999-1024, CVE-2000-1026, and subsequent ones of varying severity.

The need for systematic strengthening of TCP/IP packet parsers could only be addressed by *fully* specifying packet formats, suitable for generation of complete and comprehensive parsers. Such was the *PacketTypes* [24] proposal, which drew explicit inspiration from functional programming languages; it is the closest to our work in this respect, and also in that it suggested (but did not implement) generating the kernel's main parser for a protocol in this fashion.

In (userland) packet analyzers, *GAPA* [10] stressed the importance of a quick, intuitive description language for developing generic application level parsers. *PADS* [18], from the programming languages community, focused on resiliently parsing data expected to deviate from a "normal" protocol specification. Prior contributions include *Shield* [40] and *binpac* [31], both of which use DSLs to generate parsers for wire protocols.

Still, none of these efforts reached kernel code; ad-hoc protocol parsing remains prevalent in the kernel. The above systems, too, are intended for userland applications, independent of running kernels, such as NIDS; our goal, rather, is to protect the target's own kernel.

The other distinction between these systems and ours is that, although they all feature extensive testing, none of them aims to make the system amenable to verification. In this respect, of particular interest to us is the *Protege* system [41, 42],which uses Haskell as the basis of an embedded domain-specific language (eDSL) for describing networking protocols and generating parser code. Coming from a functional language eDSL, such code may be most amenable to verification. Unfortunately, this work and its proof-of-concept implementation of the MODBUS protocol primarily target firmware of embedded systems; as such, it doesn't support user-written policies or inclusion in mainline OS kernels.

6.3 Formal Verification & Complete Mediation

Although DSLs have long been used to produce auxiliary protocol parsers, this work has been historically separate from software verification, which generally did not tackle low-level systems code. Verifying operating system kernels (which must contain parsers so long as they contain device drivers and the lower layers of protocol stacks) remained out of reach.

The NICTA group, however, recently demonstrated the feasibility of producing a fully functional, fully verified, fully performant operating system kernel, l4.verified [22]. 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 (e.g., RockSalt [27] and Idris [11]). Recently, [33] brought together formal verification and the use of sub-Turing eDSLs in embedded firmware.

We believe that the time has come for verifiable kernel parsers for stacks that can work with other verified low-level kernel code. The worlds of DSL-based parser generation and of proving correctness of systems code should merge, and we hope that our effort contributes to this process.

7 FUTURE WORK

We anticipate producing a version of our system for Linux, and merging it with USBFilter. We also intend to work towards a formal verification of our parser, for its eventual incorporation into a verified OS kernel.

8 CONCLUSION

With our low-level USB firewall, we demonstrated that language modeling-based approach for correct kernel parsers is feasible and has strong security advantages. While other approaches leave subtle gaps or assumptions that are hard to fit into a verification lifecycle, ours is specifically designed for it. If it can be done for USB, it can be done for any complex protocol.

REFERENCES

[1] Beagle usb 12 protocol analyzer.

[2] Teensy usb development board.

[3] CVE-2011-2295, 2011.

[4] CVE-2012-3723, 2012.

[5] CVE-2013-3200, 2013.

[6] Openssl security advisory: Tls heartbeat read overrun (cve-2014-0160), 2014.

[7] Andy Davis. Lessons learned from 50 bugs: Common USB driver vulnerabilities. NCC Group publication, January 2013. https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/usb_driver_vulnerabilities_whitepaper_v2.pdf.

[8] Angel, S., Wahby, R. S., Howald, M., Leners, J. B., Spilo, M., Sun, Z., Blumberg, A. J., and Walfish, M. Defending against Malicious Peripherals with Cinch. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), pp. 397–414.

[9] Barisani, A. Forging the USB armory. In *Proceedings of the 31st Chaos Communications Conference (31c3)* (2014).

[10] Borisov, N., Brumley, D. J., Wang, H. J., Dunagan, J., Joshi, P., and Guo, C. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium* (2007).

[11] Brady, E. C. IDRIS: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages meets Program Verification* (2011).

[12] Bratus, S., Goodspeed, T., Johnson, P. C., Smith, S. W., and Speers, R. Perimeter-Crossing Buses: a New Attack Surface for Embedded Systems. In *Workshop on Embedded Systems Security (WESS 2012)* (October 2012).

[13] Cook, K. Usb avr fun, 2012.

[14] Corporation, M. User mode driver framework.

[15] David, A. Undermining Security Barriers – further adventures with USB. Infiltrate, 2012.

[16] Davis, A. Usb: Undermining security barriers. In *Proceedings of the BlackHat Technical Security Conference* (2011).

[17] Davis, A. Umap, The USB host security assessment tool. https://github.com/nccgroup/umap, 2013.

[18] Fisher, K., and Gruber, R. 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).

[19] Goodspeed, T., and Bratus, S. Facedancer USB: Exploiting the Magic School Bus. RECON.cx Computer Security Conference, June 2012. https://recon.cx/2012/schedule/events/237.en.html.

[20] Ivanov, A., Khudyakov, A., Zhuravlev, M., and Rubin, A. Darwin Nuke. https://securelist.com/blog/research/69462/darwin-nuke/, April 2015.

[21] J. Morris Jr. Types Are Not Sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1973), ACM, pp. 120–124.

[22] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)* (2009).

[23] Maskiewicz, J., Ellis, B., Mouradian, J., and Shacham, H. Mouse Trap: Exploiting Firmware Updates in USB Peripherals. In *Proceedings of the 8th Annual Workshop on Offensive Technologies (WOOT 2014)* (2014).

[24] McCann, P. J., and Chandra, S. Packet Types: Abstract Specification of Network Protocol Messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2000), SIGCOMM '00, ACM, pp. 321–333.

[25] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference* (1993).

[26] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science 17* (1978).

[27] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., and Gan, E. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2012).

[28] NCC Group and Cisco SAS team. Umap2, Second revision of NCC Group's python based USB host security assessment tool. https://github.com/nccgroup/umap2, 2016.

[29] Northcutt, S., and Novak, J. *Network Intrusion Detection (2nd Edition.* New Riders Publishing, 2002.

[30] One, A. Smashing the Stack For Fun and Profit. *Phrack 7*, 49 (November 1996).

[31] Pang, R., Paxson, V., Sommer, R., and Peterson, L. binpac: a yacc for Writing Application Protocol Parsers. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement* (2006).

[32] Paxson, V. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium* (1998).

[33] Pike, L. Hints for High-Assurance Cyber-Physical System Design. In *Proceedings of IEEE Cybersecurity Development (SecDev)* (November 2016), IEEE. Preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/sedev16.html.

[34] redpantz. The Art of Exploitation: MS IIS 7.5 Remote Heap Overflow. *Phrack 12*, 68 (April 2012).

[35] Sassaman, L., Patterson, M. L., Bratus, S., and Locasto, M. E. Security Applications of Formal Language Theory. *IEEE Systems Journal 7*, 3 (September 2013), 489–500.

[36] Tarasov, V., Zadok, E., and Shepler, S. Filebench: A flexible framework for file system benchmarking. *USENIX ;login: 41*, 1 (Spring 2016).

[37] Tian, D. J., Bates, A., and Butler, K. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)* (2015).

[38] Tian, D. J., Scaife, N., Bates, A., Butler, K., and Traynor, P. Making USB Great Again with USBFILTER. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), pp. 415–430.

[39] van Tonder, R., and Engelbrecht, H. Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation. In *8th USENIX Workshop on Offensive Technologies (WOOT)* (August 2014).

[40] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. 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).

[41] Wang, Y. *A Domain-Specific Language for Protocol Stack Implementation in Embedded Systems.* PhD thesis, ÄÜrebro University, 2011.

[42] Wang, Y., and Gaspes, V. 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).

A GENERATED CODE

As described in Section 4, our framework generates many different aspects of parser functionality from the protocol specification. Below are examples of all generated code, using the same `GET_DESCRIPTOR` message specified in Figure 5.

A.1 Generated Accessor Functions

The usability of the generatored 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 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. Whereas 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 automatically generated, they can be easily changed. For instance, one could write a function to shorten names and apply it to all identifiers simultaneously.

Generated C accessors for the `GET_DESCRIPTOR` request message are shown in Figure 11. (The duplicate "get" substring is not a typo: the first is a verb, the second is part of the noun.)

```
#define get_get_desc_req_msg_desc_type(m) (m->desc_type)
#define get_get_desc_req_msg_desc_index(m) (m->desc_index)
#define get_get_desc_req_msg_language_id(m) (m->language_id)
#define get_get_desc_req_msg_desc_length(m) (m->desc_length)
#define get_get_desc_req_msg_GET_DESCRIPTOR_data(m) (&m->data)
```

Figure 11: Generated code for accessing protocol message fields.

```
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 12: Generated code for printing protocol message contents.

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

Figure 13: Using Haskell composition to streamline development.

A.2 Generated Pretty-Printers

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 B.2.) Generated C function for legibly printing a `GET_DESCRIPTOR` request message is shown in Figure 12.

A.3 Using `withData` to Streamline Protocol Specification

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, we 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. The code in Figure 13 shows how we used this function to specify the `GET_DESCRIPTOR` response message.

Whereas the `withData` function is no doubt useful, it is most certainly specific to USB. The general lesson here is not, however, that the framework we'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.

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

B OPERATING SYSTEM INTEGRATION

We chose to integrate with FreeBSD because of its well-deserved reputation as a widely-deployed, high-performance kernel with a clean and well-documented design. We do so by means of a thin translation shim, described below.

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 section. They are intended to be as operating-system-agnostic as possible (exceptions are discussed in Section B.2).

We 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`, and 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 to show how the primitives provided by the automatically-generated code can be used to create one.

Table 7 summarizes the files involved. The primary takeaway from this table is the significant discrepancy between manually-written lines of code and automatically-generated lines of code, the latter of which are far more likely to be correct—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 the 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 B.1, we describe the method we developed for placing these hooks.

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 7: The files, both automatically 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.

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

Figure 9 shows the path by which a frame is processed by the generated verification framework.

But where are these magical, "appropriate places" whence we call fbsd_hook?

B.1 Hooks

The method for locating hooks is described below. Although applied to USB in this section, 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? Our instrumentation, described elseqhere, 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 we placed the hooks to call into the firewall. We evaluate the effectiveness of these placements in the next section, where we also discuss whether our system fulfills the requirement of complete mediation.

B.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 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 4.4, comprises the functions generated to pretty-print the content of messages. As shown in Section A.2, 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.

Our current preference, however, is to re-implement these functions to instead behave like snprintf: returning a pointer to a string instead of performing 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, handled differently by different kernels.

C USB_BB MODIFICATIONS FOR FREEBSD KERNEL

Table 8 shows the extent of modifications to FreeBSD kernel files introduced by our usb_bb DTrace probeset.

D CVE CLASSIFICATIONS

This appendix summarizes our USB-related CVE classification. For space reasons, we omit the CVEs that fall in unclear and unrelated categories and, to give examples of our reasoning, we provide only a sampling of those mitigated by policy, mitigated by design pattern, and inherently averted.

Refer to Table 6 for a quantitative summary of all vulnerabilities examined and evaluated.

D.1 Mitigated By Policy

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.

Mitigated by user policy: "reject message where length > x".

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.

Mitigated by user policy: "reject message where bNbrPorts == bad value".

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

Table 8: Enumeration of instrumented within FreeBSD's USB stack.

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.

Mitigated by 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.

Mitigated by user policy: "Reject interface_descriptor where class_id = xyz'.

D.2 Mitigated By Design Pattern

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).

Mitigated by principled buffer use 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.

Mitigated by principled buffer use 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.

Mitigated by principled buffer use would be encoded into autogeneration.

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.

Mitigated by principled data structure use encoded into autogeneration.

D.3 Inherently Averted

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.

Mitigated due to length variables 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.

Mitigated due to frame lengths automatically enforced given frame specification.

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).

Mitigated because 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 mitigated 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.