# Defending Against Malicious USB Firmware with GoodUSB

Dave (Jing) Tian
University of Florida
daveti@ufl.edu

Adam Bates
University of Florida
adammbates@ufl.edu

Kevin Butler
University of Florida
butler@ufl.edu

## ABSTRACT

USB attacks are becoming more sophisticated. Rather than using USB devices solely as a delivery mechanism for host-side exploits, attackers are targeting the USB stack itself, embedding malicious code in device firmware to covertly request additional USB interfaces, providing unacknowledged and malicious functionality that lies outside the apparent purpose of the device. This allows for attacks such as BadUSB, where a USB storage device with malicious firmware is capable of covertly acting as a keyboard as well, allowing it to inject malicious scripts into the host machine. We observe that the root cause of such attacks is that the USB Stack exposes a set of unrestricted device privileges and note that the most reliable information about a device's capabilities comes from the end user's expectation of the device's functionality. We design and implement *GoodUSB*, a mediation architecture for the Linux USB Stack. We defend against BadUSB attacks by enforcing permissions based on user expectations of device functionality. GoodUSB includes a security image component to simplify use, and a honeypot mechanism for observing suspicious USB activities. GoodUSB introduces only 5.2% performance overhead compared to the unmodified Linux USB subsystem. It is an important step forward in defending against USB attacks and towards allowing the safe deployment of USB devices in the enterprise.

## Keywords

USB, BadUSB, Linux Kernel

## 1. INTRODUCTION

The USB interface is widely acknowledged as a dangerous vector for attack. In many organizations, use of USB flash drives is restricted or outright banned [1] due to their potential for propagating malicious software. USB storage has served as a delivery mechanism by the world's most nefarious malware families [36, 38], and even in state-sponsored attacks [12]. In response, antivirus software is becoming increasingly adept at scanning USB storage for malware [26]. Recently an even more insidious form of USB-based attack has emerged. In the *BadUSB* attack [7, 25], a malicious USB device registers as multiple device types, allowing the device to take covert actions on the host. For example, a USB flash drive could register itself as both a storage device and a keyboard, enabling the ability to inject malicious scripts. This functionality is present in the *Rubber Ducky* penetration test tool [16], which is now available for public sale. Unfortunately, because USB device firmware cannot be scanned by the host, antivirus software is not positioned to detect or defend against this attack. This problem is not just limited to dubious flash drives: *any device* that communicates over USB is susceptible to this attack.

We observe that the root cause of the BadUSB attack is a lack of access control within the enumeration phase of the USB protocol. Devices are free to request that any number of device drivers be loaded on their behalf. However, existing USB security solutions, such as whitelisting individual devices by their serial number, are not adequate when considering malicious firmware that can make spurious claims about its identity during device enumeration. Standard USB devices are too simplistic to reliably authenticate, and secure devices with signed firmware that could permit authentication are rare, leaving it unclear how to defend ourselves against this new attack.

Our key insight in this work is that the most reliable source of information about a device's identity is the end user's *expectation of the device's functionality*. For example, when a user plugs in a flash drive, they are aware that they have not plugged in a keyboard. We use this insight to design and implement *GoodUSB*, a host-side defense for operating systems against BadUSB attacks. GoodUSB features a graphical interface that prompts users to describe their devices, and a kernel enforcement mechanism that denies access to features that fall outside of that description. Our system also features a security image system to simplify device administration using security pictures, and a novel USB Honeypot mechanism for profiling BadUSB attacks. GoodUSB even provides an added layer of protection for "secure" devices with signed firmware, ensuring that BadUSB attacks will still fail even if the manufacturer's signing key falls into the wrong hands.

Our contributions are summarized as follows:

- **Enforce Permissions for USB Devices**. We design and implement a permission model and mediator for the enumeration phase of the USB protocol. Our solution features an intuitive graphical interface that simplifies user participation, a Linux kernel enforcement mechanism, and a virtualized honeypot that automatically redirects and profiles potentially malicious devices. To our knowledge, our USB Honeypot is the first to appear in the literature that is capable of observing BadUSB attacks.

- **Demonstrate Robustness Against BadUSB Attacks**. We test GoodUSB against *Rubber Ducky* [16] and *Teensy* [31],

*ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA*

DOI: http://dx.doi.org/10.1145/2818000.2818040

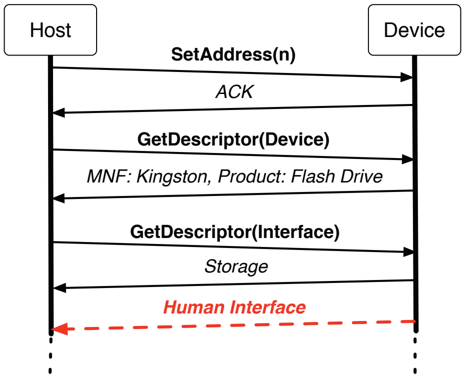


Figure 1: During USB enumeration, the host discovers the device and the drivers (interfaces) that need to be loaded in order for it to operate. In the BadUSB attack, marked by a red dotted line, the device requests additional, unexpected interfaces that allow it to perform covert activities on the system.

two of the widely available penetration and development tools that are capable of executing BadUSB attacks. We demonstrate our system's ability to block the actions of these devices' firmware. We also demonstrate GoodUSB's compatibility with a variety of benign devices, including flash drives, headsets, and smart phones.

- **Mitigate Performance Overhead**. Our results show that GoodUSB imposes only a 5.2% performance overhead (7 milliseconds) compared to enumeration in an unmodified Linux USB subsystem, with our device class identifier routine only adding 9 microseconds to the enumeration process.

Section 2 of the paper provides background on the pertinent aspects of the USB specification as well as on USB attacks. In Section 3, we identify the key challenges in securing the USB protocol against BadUSB, and propose our solutions. In Section 4 we present the full design and implementation of the GoodUSB architecture. Section 5 features our evaluation. We discuss common questions about GoodUSB, and future work in Section 6. In Section 7 we provide an overview of related work, and in Section 8 we conclude.

## 2. BACKGROUND

The Universal Serial Bus (USB) specification defines protocols and hardware used in communication between a *host* and a *device* across a serial bus [8]. In the enumeration phase of the USB protocol, a USB host controller residing within the host operating system initiates a series of queries to discover information about the device's functionality. A simplified example of device enumeration for a USB flash drive is shown in Figure 1. After enumeration, the host loads USB kernel drivers (*Interfaces*) that allow the device to operate; USB devices are often complex, serving multiple functions, and therefore a single device can request one to many interfaces from the host. While some of the more sophisticated USB devices, notably smart phones, often require a custom interface, there are also hundreds of single-purpose interfaces that are defined in the specification [8]. For compatibility reasons, most USB devices usually use these standard interfaces whenever possible.

### 2.1 When USB Goes Bad

USB devices are widely acknowledged as a dangerous vector for attacks, particular with regards to storage devices. Flash drives not only permit the exfiltration of sensitive data, they also facilitate the propagation of viruses and other malware. Due to the popularity of auto-run features in operating systems, malicious payloads carried on USB storage can at times be installed without user knowledge or consent. This makes flash drives particularly devastating when used in social engineering attacks, where the compromise of a host can be as simple as tricking the user into plugging in a USB key. As a result of these threats, enterprises often restrict or ban the use of USB storage devices, even going so far as to super glue USB ports shut [1], depriving employees of the benefits of portable storage media.

Recently, a more insidious form of USB-based attack has emerged – rather than placing malicious code within storage, malware is embedded into the firmware itself. In a class of attacks broadly referred to as *BadUSB*, malicious devices covertly request additional interfaces during enumeration, allowing them to attack the host. An example of Nohl et al.'s BadUSB attack is denoted by the dotted line in Figure 1, in which the firmware of a flash storage device is rewritten to register as both a storage and human interface device (HID), allowing it to inject keystrokes that open a shell and download malware from the Internet [7, 25]. Similar functionality is provided by the Rubber Ducky penetration test tool and the Teensy USB development board, which are sold online [16, 31]. Unlike traditional USB attacks, in BadUSB, the host cannot use antivirus software to detect the presence of a malicious payload within the device. Moreover, with BadUSB, *any* device is a potential attack vector, not just storage devices.

The use of signed firmware as a defense against BadUSB is noted by both Nohl et al. [25] as well as Imation's IronKey team [19]. Indeed, signed firmware dramatically increases the complexity of performing a BadUSB attack. This is because sophisticated USB devices, such as IronKey, are able to measure device firmware and verify its signature before permitting it to load. However, signed firmware is not a panacea against powerful state-sponsored adversaries, who through coercion or outright attack may be able to obtain device manufacturers' signing keys. In fact, obtaining legitimate signatures for malicious device drivers was an integral step in the success of the Stuxnet attack [12]. Even when considering "secure" USB devices, there is still a need for a defensive layer to protect from BadUSB attacks.

## 3. DESIGN

In this section, we identify the key challenges to the design of a security mechanism for USB enumeration. In considering BadUSB, we observe that the root cause of this threat is that USB drivers effectively represent a set of system privileges, and yet the USB protocol does not provide a means of restricting devices' access to these privileges. Therefore, solving the BadUSB problem requires the introduction of a security layer to the enumeration phase of the USB protocol. We discover that a number of unique challenges exist due to the plug-and-play nature of USB that prevents traditional access control mechanisms from being suitable in this environment, and subsequently propose solutions to each of these obstacles. The technical details of our solution can be found in Section 4.

### 3.1 Threat Model & Assumptions

The goal of our system is to provide a defense against BadUSB attacks in a security-conscious enterprise environment where myriad different USB devices are used each day. Due to the sensitive nature of such organizations, they have already deployed advanced USB malware scanning kiosks that effectively detect mali-

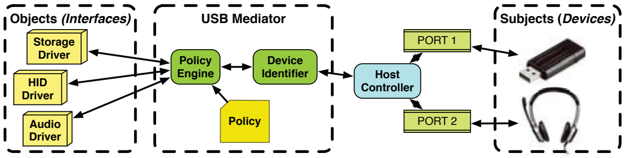


Figure 2: GoodUSB introduces a mediator in the USB stack of the host. The mediator restricts USB devices (subjects) access to USB drivers (objects) according to policy.

cious storage payloads [26, 28]; hence, traditional USB attacks are not a concern. We assume the use of standard commodity devices that lack advanced security features such as signed firmware. While signed firmware can be employed to defeat BadUSB, such features are costly, and to our knowledge are only available for USB storage devices [18]. We assume that employees in our operating environment are required to participate in a security orientation.

We consider an Advanced Persistent Threat attack that is attempting to further its presence in the enterprise through distributing USB devices with malicious firmware (i.e., BadUSB). The malicious devices have entered the physical premises via supply-chain compromise or social engineering. We conservatively assume that these devices are subject to byzantine faults during participation in the USB protocol. The device may make any claim about its identity during enumeration, and can attempt to confuse or evade the device identification mechanism that our system introduces; for example, the device can lie about its manufacturer and product ID. The device may also alter its responses each time it enumerates. Moreover, the adversary may have changed the physical casing of the device so that its functionality is not apparent through visible inspection.

Finally, we make the following assumptions about the state of the host system on which our security mechanism is being deployed. We assume the host is in a correct state prior to connecting to any USB devices. We also assume that the host's USB software stack is correct, and does not contain any exploitable software flaws. Conceivably, a BadUSB device could send malformed messages that could exploit a software vulnerability (e.g., buffer overflow) in the host controller or driver. This is an important problem in itself, and fuzzing techniques have been proposed elsewhere in the literature to detect such faults [35]; however, it is orthogonal to our goal of addressing a fundamental vulnerability in the USB protocol.

## 3.2 Mediating USB Interfaces & Drivers

The fundamental vulnerability in USB that gives rise to BadUSB attacks is that arbitrary USB interfaces can be enumerated, comprising a set of unrestricted privileges provided to a USB device. In response, we propose the introduction of a permission validation mechanism that authorizes device's access requests to individual USB interfaces. The proposed mediator is shown at a high level in Figure 2. During the USB enumeration phase, a **Device Identifier** authenticates the connected device and provides an subject ID. When the host queries the device for its interfaces, the device's response represents an *access request*. The subject ID and access request are passed into a **Policy Engine**. Based on the **Policy**, the engine then individually authorizes the requested interfaces prior to loading the drivers on behalf of the device.

While restricting device activity at the driver granularity is a significant improvement over the status quo, it would also be desirable to restrict device actions at finer granularities. For example, a device that registers as a Human Interface Device (HID) but only makes use of the *Volume Up* and *Volume Down* keys is dangerously over-capable; while use of those two keys alone is harmless, with the full HID driver the device can effectively take any action on the host. Unfortunately, this requires instrumentation of individual USB drivers, so it is not a general solution to the BadUSB problem. However, our mediator must be extensible, supporting security-enhanced USB drivers as they are made available. In Section 4, we instrument the general USB HID driver to provide access to volume controls only, preventing a USB headset from running in an over-capable state, such as running as a keyboard.

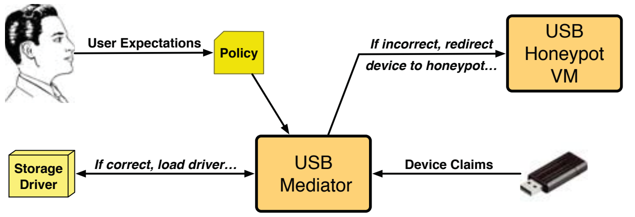


Figure 3: GoodUSB cannot trust what the device claims to be during enumeration; however, the device's claims can be verified by checking them against the user's expectation as to what the device is and how it should operate. If verification fails, the device is flagged as potentially malicious, and is redirected to a honeypot virtual machine.

## 3.3 Identifying USB devices

We now describe the **Device Identifier** component of our USB mediator. A fundamental requirement of any access control system is authenticating the subject. The device descriptor passed by the USB device during enumeration contains information such as the manufacturer, product, and a unique serial number for the device. However, the problem of identifying the device is actually much more complicated. As we assume devices are subject to byzantine faults, we cannot trust any message that we receive from the device during enumeration. If an adversary has rewritten a device's firmware, it can change its response during any message in the enumeration, including lying about its manufacturer and model number. When the device's reported descriptor and even its physical appearance are potentially false, how can we identify the device?

We assert that the most reliable source of information of a device's identity is the end user's expectation of the device's functionality. The purposes of most USB interfaces are intuitive, especially in an environment where all users are computer literate and have been instructed on security procedures. We propose that the simplest means of enforcing least privilege on USB devices is to ask the users what they expect their device to do, having the user serve as a *verifier* for the claims that the device makes during enumeration. This verification concept is visualized in Figure 3. When a device first connects to the host, the user is notified of the connection through a graphical dialog box on the host. The dialog prompts the user to select the features (i.e., interfaces) that they wish to enable on the device. Note that, since the host is trusted, this constitutes a trusted path from the host controller to the user. The user's settings are stored in a **policy** database, with each record being a tuple <*Subject ID, Authorized Interfaces*>. The subject ID includes *Manufacture*, *Product*, *Serial Number*, etc.[1]

On subsequent connections, one of three scenarios may occur:

[1] The record format and subject ID are simplified here for better illustration.

1. The device's claim is consistent with the prior connection.
2. The device makes a different claim that *matches* an entry in the device database.
3. The device makes a different claim that *does not match* an entry in the device database.

In Scenarios (1) and (2), the user will be presented with an entry from the database and asked whether the information is correct. In Case (1), the user confirms that the information is correct, and enumeration is permitted to continue for the authorized interfaces. In Case (2), the user reports that the information is incorrect, and the device is flagged as potentially malicious. In Case (3), the user is presented with the initial device registration dialog again. However, since the user knows that the device has already been registered, he or she can report the anomaly and the device will be flagged as potentially malicious.

Given the above description, readers may find themselves understandably wary of the burden that our system places on the end user. However, given the capabilities of our attacker, and the lack of a core root of trust for measuring USB firmware, we assert that our solution is the *only* option for deterministically authenticating devices, which is a prerequisite to defending against BadUSB. In Section 4, we present the technical details of the GoodUSB graphical interface, which makes use of visual cues to dramatically simplify the process of administration for normal users with limited technical background.

## 3.4 Profiling Malicious USB Devices

Once a device has been flagged as potentially malicious, what actions can we take? Unfortunately, it is not possible to block the device from further interactions with the system. On subsequent connections, the device can make different claims about its identity, so we have no means of blacklisting it. We determined the most valuable action that our system could take is to redirect the device to a virtualized honeypot, allowing the device to be observed while simultaneously protecting the host.

Within the virtual honeypot, the actions taken by the device can be profiled, which could prove valuable in the ensuing forensic investigation. The honeypot's interactions with other system components is shown in Figure 3. We identify the following types of information as valuable to an investigator: device information after enumeration, device drivers loaded for the device, and all communication at the USB layer, including *all keystrokes* and *all IP packets sent/received over the network*. This information could potentially be passed to high-level forensic tools for detailed inspection and an intrusion detection system (IDS) which would provide a means of remediation in the event that a device is incorrectly flagged as malicious.

# 4. GOODUSB IMPLEMENTATION

In this section, we present GoodUSB, our fully-implemented security architecture for the Linux USB stack. While our BadUSB defense is general enough to apply to any operating system, we have implemented GoodUSB for Ubuntu 14.04 LTS (kernel version 3.13.11). The full architecture of GoodUSB, shown in Figure 4, introduces four components. First, a user space daemon handles the graphical interface and policy management, and also includes the logic for the USB mediator. A second user space component, a KVM honeypot, profiles potentially malicious USB devices. In kernel space, we introduce a device class identifier, as well as a limited USB HID driver that secures human interface devices by restraining them to particular kinds of keystrokes. The kernel hub thread is minimally modified to interoperate with the components of the GoodUSB architecture.

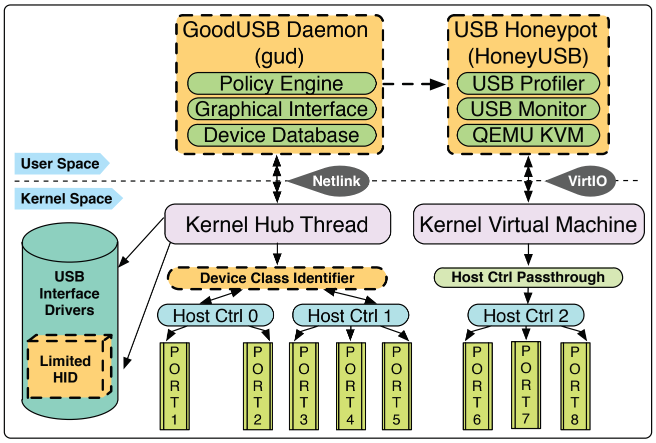


Figure 4: The GoodUSB architecture. Components that are introduced by GoodUSB are colored orange and bordered by dashed lines. The kernel hub thread is also modified to interoperate with our system components.

## 4.1 User Space Daemon

Most of GoodUSB's functions are handled by a user space daemon (a.k.a. gud). Shown in Figure 4, gud includes three subsystems: a *policy engine* that implements the USB mediator logic, a *graphical interface* that features a security picture recognition system, and a device database that associates a device's claimed identity and functionality with the user's expectation of the device. To allow gud to interact with the rest of the USB stack, we use the new netlink socket created in the kernel hub thread (a.k.a. khub), which communicates with gud to perform USB device detection, enumeration and driver matching/loading. The subsystems of gud are detailed below.

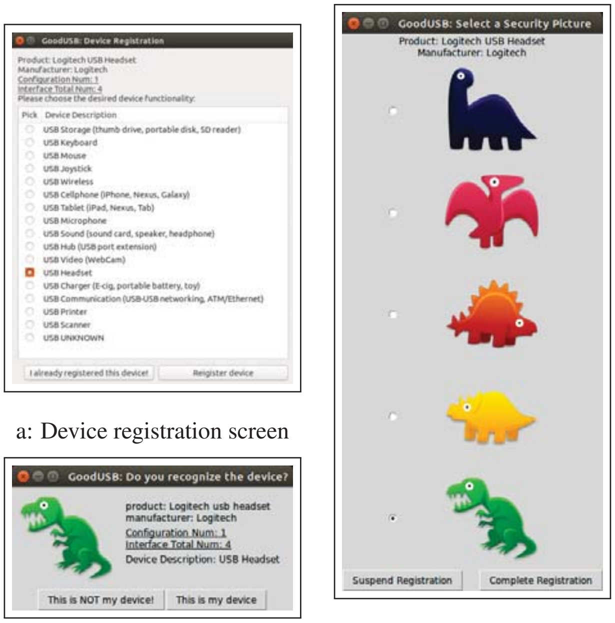
**Policy Engine.** The policy engine is responsible for determining whether the requested interfaces of a newly connected USB device match the user's expectation of device functionality, and subsequently enforcing that expectation. It notifies the kernel space components to block the loading of particular interfaces, or to redirect the device to a honeypot when it is potentially malicious.

To increase the users' experience, the policy engine maintains a mapping between low-level interface types and a high-level summary of common USB devices. Some example mappings are shown below:[2]

```
USB_DEV_STORAGE=>          USB_CLASS_MASS_STORAGE
                           USB_CLASS_CSCID
                           USB_CLASS_VENDOR_SPEC
USB_DEV_CELLPHONE=>        USB_CLASS_MASS_STORAGE
                           USB_CLASS_VENDOR_SPEC
USB_DEV_HEADSET=>          USB_CLASS_AUDIO
                           USB_CLASS_HID (LIMITED)
                           USB_CLASS_VENDOR_SPEC
USB_DEV_CHARGER=>          {0}
```

Reading the above mappings, the policy states that storage devices can only register the following interfaces: MASS_STORAGE (for flash drives), CSCID (for smart cards) and/or VENDOR_SPEC interfaces. Storage devices *cannot* register the HID interface, preventing the most widely recognized form of BadUSB attacks. In addition to the AUDIO interface, certain USB headsets sometimes

[2] There are 17 mappings in total. Only 4 are presented here.



GoodUSB: Device Registration

Product: Logitech USB Headset
Manufacturer: Logitech
Configuration Num: 1
Interface Total Num: 4
Please choose the desired device functionality:

| Pick | Device Description |
|---|---|
| ○ | USB Storage (thumb drive, portable disk, SD reader) |
| ○ | USB Keyboard |
| ○ | USB Mouse |
| ○ | USB Joystick |
| ○ | USB Wireless |
| ○ | USB Cellphone (iPhone, Nexus, Galaxy) |
| ○ | USB Tablet (iPad, Nexus, Tab) |
| ○ | USB Microphone |
| ○ | USB Sound (sound card, speaker, headphone) |
| ○ | USB Hub (USB port extension) |
| ○ | USB Video (WebCam) |
| ● | USB Headset |
| ○ | USB Charger (E-cig, portable battery, toy) |
| ○ | USB Communication (USB-USB networking, ATM/Ethernet) |
| ○ | USB Printer |
| ○ | USB Scanner |
| ○ | USB UNKNOWN |

a: Device registration screen

GoodUSB: Do you recognize the device?

product: Logitech usb headset
manufacturer: Logitech
Configuration Num: 1
Interface Total Num: 4
Device Description: USB Headset

GoodUSB: Select a Security Picture

Product: Logitech USB Headset
Manufacturer: Logitech

○
○
○
○
●

b: Recognized device notification c: Security picture screen

Figure 5: Screenshots from GoodUSB user interface.

require the HID interface for volume control. GoodUSB introduces a limited HID interface that restricts the permissible keystrokes of non-keyboard USB devices, which prevents malicious HID devices from taking control of the system. Another interesting example is a CHARGER device, which does not contain any interfaces. As a matter of fact, these chargers should never be detected as USB devices, because the charging procedure does not need to involve any USB layer communication. Through enforcing this permission mapping, GoodUSB is able to defend against BadUSB attacks. Note that vendor specific interfaces are allowed in most devices. This is a tradeoff between security and usability, as devices that require a vendor specific driver are likely to break if denied this interface. We discuss this limitation in Section 6.

Based on GoodUSB's configuration, the policy engine can operate in either *basic* or *advanced* modes. In basic mode, the graphical interface features high-level device summaries, as shown in Figure 5a, and the user selects a single option that maps to low-level interfaces. In advanced mode, the graphical interface instead shows the low-level interfaces, and allows the user to make multiple selections. The advanced mode allows the user to exercise finer control over device functionalities, and also supports devices that require uncommon interface sets.

**Graphical Interface.** When a USB device connects to the host, the policy engine loads one of several dialog boxes depending on whether the device is recognized from a previous session. If a device is not recognized, GoodUSB prompts the user with the device registration box shown in Figure 5a. [3] The text field at the top of the box allows the user to confirm that the device's claimed identity (i.e., Manufacturer and Product) match the device that was just plugged into the host. The remainder of the box provides a set of device descriptions for the user to select. Each device description maps to a set of permissible interfaces.

Immediately after the device registration screen, the user is asked to select a security image to associate with the device, as shown in Figure 5c. Security images are widely used as an anti-phishing mechanism by banking websites [21]. In GoodUSB, the security image component is introduced to simplify device administration, and also provides a visual cue for the presence of a potential BadUSB attack. Recall that BadUSB devices can spoof any message in their device descriptors; an adversary who is aware of our GoodUSB defense may therefore attempt to masquerade as a known device that has the desired interface, e.g., the HID interface.

When GoodUSB recognizes a device, it is either a legitimate occurrence or evidence of an attack. The dialog box for recognized devices is shown in Figure 5b. Here, the option for selecting a device type has been removed. The user can verify the device through either reading the descriptive text or checking that the presented security image is correctly associated with the device. If the presented information is incorrect, the device is flagged as potentially malicious and is redirected to the USB honeypot. Otherwise the user approves the device and driver loading continues.

**Device Database.** Once gud obtains user expectations and a security picture is selected, this information is recorded alongside the output of the Device Class Identifier in a database. The database is implemented as a binary file, and is synchronized with kernel space whenever a new USB device is plugged in. When the machine is rebooted, gud re-transmits the device database to kernel space via the netlink socket, making sure that previously classified devices will be recognized on subsequent connections. If needed, users can also clear the database in gud, which provides a clean base in the kernel space as well, once the machine is restarted.

## 4.2 USB Honeypot

In the event of a potential attack, administrators will undertake forensic investigation to determine the nature of the attack and identify likely culprits. To observe the activities of potentially malicious devices, GoodUSB features a honeypot virtual machine mechanism. While honeypots for malicious USB devices have been previously proposed by Poeplau and Gassen [32], we realized that these systems are actually incapable of observing the BadUSB attack vector. The reason is that their system emulates a *device*, as opposed to a *host*, and attempts to catch host-based malware as it infects the device. BadUSB is not a host-to-device attack, but rather a device-to-host attack. In BadUSB, once the host is compromised, the adversary will have to rely on other attack vectors to extend their presence in the devices, as it is very difficult (if not impossible) to infect the firmware of USB devices simply by having them connect to an infected host. Therefore, it is necessary to design a new USB honeypot framework that is suited to observing BadUSB.

Our system, *HoneyUSB*, is a QEMU-KVM virtualized Linux machine containing multiple device profiling services. HoneyUSB supports two modes of device observation/profiling. In the first, HoneyUSB reserves an entire USB controller device on the host, and the host controller device (HCD) is hoisted directly into KVM using pass-through technology. The advantage of this profiling mode is that the potentially infected device never operates directly within the host OS, and is effectively physically separated from the host machine. Using this profiling mode is helpful when out-of-band knowledge has been used to flag a device as potentially malicious, e.g., it was found lying in the company parking lot. In a second mode, gud automatically redirects devices to HoneyUSB after the user flags them as potentially malicious.

The honeypot VM, which also runs Ubuntu Linux, is preconfigured as follows. We enabled usbmon in the VM's kernel [42], which acts as a general USB layer monitor, capturing all the USB packets transmitted by the device. In the user space, we created a

[3] A user study of the prompt's effectiveness is out of the scope of the paper.

USB profiling application, usbpro, which aggregates device information from sysfs, lsusb, usb-devices and device activities from usbmon and tcpdump. Moreover, a new udev rule with high priority is associated with usbpro, guaranteeing that usbpro is loaded prior to device enumeration. Thus, the report generated by usbpro is an exhaustive description of the device's reported information as well as the actions taken by its associated drivers. Excerpts of a report for a HID device generated by usbpro can be found in the evaluation section.

HoneyUSB also contains an instrumented version of the GIO Virtual Filesystem (gvfs), the user-space driver used by USB-enabled cellphones such as Android. We have extended gvfs to collect file-level data provenance, constituting a detailed description of the read and write operations performed by the device. Currently, file-level provenance has been added into the MTP backend to support Android, which means usbpro is able to record all the file-based I/O operations happened in the Android phone operated in MTP mode.

## 4.3 Device Class Identifier

The Device Class Identifier is a kernel space component that summarizes the claims made by the device during enumeration. This summary contains both the device descriptor fields that are presented to the user in Figure 5, and all the descriptors transmitted by the device during enumeration, plus the current active configuration. This includes the device information and requested interfaces in the active configuration, and also other configuration supported but not used by the device. A SHA1 digest is then computed based on the summary [4].

The digest is used as a device identifier in both the gud device database, allowing gud to recognize devices that were previously registered, and the kernel device database, keeping it synchronized with the former. After the USB enumeration, the kernel knows all the interfaces requested by the device, as well as the SHA1 digest. If the digest does not match a prior entry from the kernel device database, the kernel notifies gud to present the device registration screen to the user (Fig. 5a). The user's response is transmitted to the kernel by gud before the requested interface drivers are loaded. After receiving instructions from gud, the kernel first creates an entry for the newly registered device in its device database, if the device is to be enabled. The permitted drivers are then loaded, while other requested drivers are ignored, thus ensuring that the device cannot interact with the system in ways that were not expected by the user.

When there is a match in the kernel device database, meaning this device is recognized as a known one, the kernel notifies gud and asks for permission to enable this device (Fig. 5b). The requested drivers are not loaded until after the user has approved the device. If the user disapproves the device, the kernel disallows any interfaces requested by the device by not loading any drivers and gud helps redirect the device into the USB honeypot.

## 4.4 Limited HID Driver

The GoodUSB architecture is designed primarily to enforce least privilege on USB device at the granularity of device drivers. Unfortunately, devices that have been approved for a particular interface are free to operate with the full capabilities of the associated driver. In Section 3.2, we mentioned that this is particularly troubling for the the Human Interface Device (HID) interface, which allows a BadUSB device to take nearly any action on the system [25]. While GoodUSB allows users to disable the HID interface of the USB device completely, there are cases where the HID interface is legal and a functioning part of the device, e.g., the HID interface of a USB headset controlling the volume of the internal speaker.

```
usbpro HID analyzer started:
===========================
_F2__x_t_erm_ENTER
_p_w_d_ENTER
_i_d_ENTER
_c_a_t_SPACE/etc/passwd_ENTER
_
===========================
usbpro HID analyzer done
```

Figure 6: GoodUSB's profiling tool, HoneyUSB, captures injected keystrokes from a USB storage device maliciously exposing a keyboard (HID) interface.

To mitigate the danger of HID-based BadUSB attacks, we have instrumented a copy of the Linux USB HID driver to restrict the number of characters that can be injected by USB devices. The Linux USB HID driver is widely used by many USB devices because it bridges the USB and Input layers in the kernel. As USB Requests Blocks (URBs) are a common abstraction for USB packets within the kernel, we instrumented the USB HID driver at the URB level, which saved us from having to perform packet inspection. We modified the original USB HID driver to restrict the kinds of URBs the driver can report to the higher-level input driver. The current limited USB HID driver supports only 3 different keystrokes, corresponding to volume increase, volume decrease, and the mute button, as are commonly found on USB headsets.

Exercising control of device activity above the interface level requires instrumenting the various USB drivers to support access control, like what grsecurity [27] does, which is tedious, potentially error-prone and volatile to driver changes and new drivers. However, our limited USB HID driver demonstrates that our approach can dramatically reduce the scope of BadUSB attacks by limiting the general USB HID driver without touching any specific drivers.

# 5. EVALUATION

We now evaluate the GoodUSB architecture. We first provide a security case study in Section 5.1, where GoodUSB is tested against a variety of malicious and benign devices. In Section 5.2, we provide a performance evaluation of our system.

## 5.1 Attack Analysis

The authors of BadUSB have published a proof-of-concept implementation online [7] with reverse engineered firmware for a particular USB storage device that adds a malicious HID interface. Rather than use this highly specific instance of BadUSB, we use several popular penetration and development tools to launch a variety of attacks in order to demonstrate the range of defenses provided by GoodUSB.

### 5.1.1 HID-Based Attacks

To demonstrate GoodUSB's resistance to attacks from exposing human interface device (HID) interfaces (e.g., exposing keyboard functionality), we use the Rubber Ducky penetration testing device. The Ducky provides a user-friendly scripting language enabling different HID-based attacks. We load a basic Ubuntu terminal command script [17] into the Ducky, which opens an xterm window once the Ducky is plugged into the victim's computer. It then is-

[4] We assume that the digest could be forged in our threat model and we allow that happen in GoodUSB. Therefore, there is no need to pursue the best secure hashing function and SHA1 is usually optimized for better performance in the kernel.

sues several commands, including checking the /etc/passwd file. The first time we plug in the Ducky, GoodUSB pops up the device registration GUI, asking users for their expectations of the device's functionality. Since the Ducky appears to be a USB thumb drive, we choose "USB Storage" and register it with a security image selected from a list, as shown in Figure 4c. The attack fails because GoodUSB does not allow USB HID interfaces for "USB Storage" devices. However, the Ducky continues to function in its capacity as a storage device.

When the Ducky is plugged in again, GoodUSB recognizes it and shows the correct security picture. Rather than enabling this device as a USB storage device, we click "This is NOT my device!", which redirects the Ducky into HoneyUSB. Using the usbpro utility, we can easily see all the information and activities of the ducky, including reconstructing its injected keystrokes, as shown in Figure 6.

### 5.1.2 *Other USB Interfaces and Composite Devices*

We demonstrate more robust interface attacks using a Teensy USB development board [20]. Unlike the Rubber Ducky, Teensy is able to simulate not only USB HID devices but also USB Serial devices, USB MIDI devices and others. Moreover, Teensy is also able to combine different interfaces together to make a composite device, which is how devices such as USB headsets and smartphones present themselves to hosts.

First, we consider a scenario where a Teensy presents a USB storage form factor but is acting as a serial device to transmit messages (e.g., shell scripts) to a trojan residing on the host machine. To accomplish this, we program a Teensy 3.1 board to expose a serial terminal at /dev/ttyACM0. When the board is plugged in, it attempts to communicate over the serial interface to the trojan listening on the tty interface. Based on its form factor, however, the user registers the Teensy as a USB storage device with GoodUSB. Consequently, the serial interface is not exposed and the trojan cannot receive its commands.

We next use the Teensy to demonstrate GoodUSB's ability to handle composite devices. We program the Teensy to simultaneously register itself as a keyboard, a joystick, a mouse and a serial port. Each interface is controlled by a separate task on the board; for example, one job instructs the mouse to move around the screen, while an independent task controls the joystick. With the help of the advanced mode of gud, GoodUSB displays all the interfaces requested by the Teensy before any drivers are loaded. This allows the user to whitelist individual interfaces; for example, we can enable mouse functionality while disabling all other input types. The result is that GoodUSB is able to enforce least privilege over the composite device by disabling other undesired functionalities requested by the device.

### 5.1.3 *Smartphone-Based USB Attacks*

The authors of BadUSB released a shell script called *BadAndroid* that emulates a DNS-based Man in the Middle (MitM) attack on the host machine using a rooted Android phone[5]. The basic functionality required by this attack is USB Tethering, which allows a USB device to present itself as an Ethernet card to the host. In this experiment, we connect a Nexus S phone to GoodUSB and register it as "USB Cellphone." GoodUSB only permits the smartphone to use the mass storage and vendor-specific interfaces. At first, Nexus S only registers the storage interface. However, when we enable USB Tethering on the phone, GoodUSB detects the new interface request and pops up the registration window again, asking for the user's permission. Only if the user explicitly selects "USB Cellphone with Tethering" will the network interface be available. If the standard "USB Cellphone" description is again selected, tethering over USB, and the potential DNS MitM attack, is thwarted.

| Action | Min | Avg | Max | Mdev | Overhead |
|---|---|---|---|---|---|
| Normal Enumeration | 140266 | 140424 | 141001 | 126 | N/A |
| GoodUSB Steps: | | | | | |
| Device Identification | 8.0 | 9.0 | 10.0 | 0.2 | N/A |
| First Enumeration | 146308 | 147675 | 149336 | 609 | 5.2% |
| Second Enumeration | 146306 | 147463 | 149268 | 558 | 5.0% |
| Honeypot Redirect | 248951 | 262057 | 295444 | 6842 | N/A |

Table 1: Microbenchmarking GoodUSB operation (in microseconds) averaged over 20 runs.

Alternately, when GoodUSB presented a second device registration window, we could have flagged the device as potentially malicious. The Nexus S would have then been redirected to HoneyUSB and been granted permission to register the USB Tethering interface, where usbpro would have observed the IP packets sent and received by the phone. If there is a legitimate need for additional interfaces such as the network interface for tethering, GoodUSB can provide this support through the advanced interface menu or through adding an additional device-to-interface mapping (e.g., tethering-enabled phones) on the basic menu.

## 5.2 Performance Analysis

The utility of GoodUSB depends on its imposing minimal overhead on the host. Below, we provide a micro benchmark based on the different operations of GoodUSB. Our host machine is a Lenovo ThinkCentre desktop, with a 3GHz Intel(R) Core(TM)2 Duo CPU (2 cores) and 4 GB of RAM. HoneyUSB, which executes inside a KVM virtual guest, runs on the same host, with 2 virtual CPUs and 2GB memory. Both are running Ubuntu Linux 14.04 LTS with kernel 3.13. The testing USB device is a Logitech ClearChat USB headset H390, containing 4 interfaces (3 audio + 1 HID). To precisely measure the overhead imposed by the core system rather than user interactions, we bypass the measurement of the GUI component by hard-coding messages to the kernel from the user daemon. All measurements are based on 20 enumerations using same device plugged into the same USB port on the test machine.

Table 1 provides the results of our measurements. *Normal Enumeration* displays the time required to add a new USB device by the original khub thread in the kernel without GoodUSB enabled. *Device Identification* shows the overhead of our device class identifier, which measures all the descriptors from the USB device and the current configuration using SHA1. The average overhead for this step 9 $us$, which is almost negligible compared to the whole USB enumeration, which takes about 140 $ms$. *First Enumeration* demonstrates the case when GoodUSB is enabled where the device is plugged in for the first time (within the user space, both the device registration and the security picture selection GUIs would be popped up). Compared to the original device adding procedure, GoodUSB only introduces 5.2% overhead. *Second Enumeration* shows the case where the device is recognized by GoodUSB (within the user space, only the device recognition GUI would show up). Compared with the original procedure, GoodUSB only presents 5.0% overhead. Finally, we measure the overhead of HoneyUSB redirection in *Honeypot Redirect*. Note that HoneyUSB is already started in our evaluation and it usually takes 5–10 seconds to start it in our host machine. Once HoneyUSB is running, the whole redi-

[5] The malicious phone sends the host false DNS information, e.g., the IP address of an attacker-controlled server for a banking web site to steal credit card information.

rection needs only 262 $ms$ to allow the device to re-enumerate.

We performed similar tests with a Kingston 2GB USB thumb drive and a Nexus S phone with/without GoodUSB. The enumeration times are comparable - the overhead is 5.1% for the USB storage and 7.3% for the phone. The phone appears to have larger overhead because it enumerates more quickly in our testing, which is 2275 $us$ in average without GoodUSB enabled. Because USB is a master-slave protocol, the device's ability to modify the speed of enumeration is limited. The speed is dictated by USB 2.0 bus speeds and the processing delay on the host. The enumeration of a headset is slower than a flash drive because it is registering more interfaces, which causes more processing on the host and more data to be sent over the USB interface. GoodUSB's overhead is thus virtually negligible during the USB device enumeration phase. There is no impact at all on regular device operations (e.g., file transfer, mouse movement, etc.) after the enumeration phase.

# 6. DISCUSSION

*Does selectively disabling interfaces break USB devices?*

We tested GoodUSB against a number of devices found in our laboratory and commonly used, including USB keyboards, mouses, flash drives, headsets, wireless adaptors, webcams, smart phones and chargers, and can anecdotally report that selective authorization of interfaces does not prevent benign USB devices from performing the authorized functions in most cases. For example, we tested GoodUSB against a Logitech USB headset that requested interfaces for Audio (Input), Audio (Output), and Human Interface Device (HID). Each feature was able to work in isolation when the others were disabled, e.g., a headset with HID interface disabled. This is an exciting potential application for GoodUSB, as some enterprise environments may wish to selective disable certain features (e.g., the microphone found in a headset) for fear that they be misused by malware. We expect that compatibility issues will arise in instances where USB device developers make unexpected use of interfaces. One example may be the yubikey [41], an authentication aid that is both a USB smart card and a HID keyboard. While there always exist some USB quirks and a serious USB device survey is needed to tell how diverse the combination of interfaces is, for these unusual cases, GoodUSB can be easily extended to support these special devices by adding new device-to-interface mappings.

*Can GoodUSB authenticate individual USB device units?*

Because devices can lie about their identity, GoodUSB relaxes the concept of authentication, instead seeking to identify *classes* of devices at the granularity of the product type under the same manufacturer. This is sufficient for the goals of our system, which only seeks to restrict the interface set available to certain kinds of devices; all USB devices of the same model should require access to the same interfaces.

*Can GoodUSB protect against malicious smart phones?*

Smart phones are troublesome for GoodUSB due to the use of the vendor-specific interface. To minimize compatibility issues, GoodUSB allows the vendor-specific interface to be loaded for most kinds of devices. Many smart phones, including Android and iPhone devices, request the vendor-specific interface during enumeration. Because the phone's actions are ultimately dependent on a user space driver, GoodUSB cannot make a determination as to the device's potential actions until after the device has loaded. To provide some confidence as to the intent of the device, we recommend plugging smart phones into HoneyUSB via passthrough, where `usbpro` is able to profile the phone in the sandbox.

To demonstrate, we profiled the Nexus 5 and iPhone 3GS in HoneyUSB. `usbpro` reported that Nexus 5 used the vendor-specific interface to load the `usbfs` kernel driver. Different from other USB kernel drivers, the only functionality provided by `usbfs` is to expose the device node to user space on the host and to enable file I/O operations. From there, Nexus 5 loads `gvfsd-mtp` to perform the Media Transfer Protocol (MTP) over USB connections. The iPhone 3GS uses two vendor-specific interfaces for loading the `usbfs` and `ipheth` kernel drivers. In user space, the `usbmuxd` driver allows data synchronization between the host machine and iOS device. This serves to demonstrate that HoneyUSB can be used independently of the rest of our system to profile potentially malicious smart phones.

*Can GoodUSB be used as IDS?*

Though GoodUSB rests on the final decision of users, it is possible to extend GoodUSB into an IDS for USB devcies, assisting users to identify anomalous combination of interfaces (storage + keyboard), as well as anomalous device behaviors (delayed registration of a keyboard, for instance). For the former, GoodUSB can pop up an warning window, notifying the policy violation of the device to the user, rather than disabling the anomalous interface silently. For the later, `usbpro` could be used to learn the normal behaviors of devices, and to detect abnormal behaviors in the future. Machine learning based techniques using timing side channels [2] can also be integrated into GoodUSB, helping detect abnormal behaviors of devices early in the USB enumeration phase.

*Is GoodUSB easily deployable?*

GoodUSB was designed with consideration for users with limited technical knowledge. The GoodUSB daemon provides a basic mode to abstract away low-level interface decisions, simplifying device administration for regular users. Additionally, the daemon's security image component speeds up the process of authorizing devices on subsequent connections.

One obstacle to the deployment of GoodUSB is the requirement of a custom kernel. Instrumenting the kernel is necessary to introduce a security mechanism into the USB stack. To ease the installation and configuration of GoodUSB, we will be releasing GoodUSB in multiple formats upon publication. In addition to a kernel patch, we will also publish a prebuilt x86-64 GoodUSB kernel image for Ubuntu Linux users. Additionally, we will provide a preconfigured GoodUSB KVM image, as well as a separate KVM image for HoneyUSB, in order to make deployment of GoodUSB feasible and straightforward.

## 6.1 Future Work

GoodUSB is a first step in hardening the USB stack from sophisticated attacks. In future work, we intend to move up the stack to explore USB drivers. While best practices in software engineering encourage drivers to support as many devices as possible, this inherently violates the principle of least privilege, providing a device with more abilities than it actually needs. We plan to perform a driver analysis that explores this problem in depth. We also intend to analyze some of the more popular user space drivers, such as `usbmuxd`, and instrument them to provide file-level provenance so their actions on the system may be better understood.

We also hope to add more features to the GoodUSB architecture. One such feature is to let the profiling phase of HoneyUSB inform the available device-to-interface mappings in the `gud` graphical interface, thereby automating the process of adding new mappings to the policy engine. We also hope to use HoneyUSB profiles to improve GoodUSB's ability to predict the purpose of vendor-specific interface requests, allowing `gud` to display the actual expected driver to be loaded to the user.

## 7. RELATED WORK

Awareness of the USB attack vector has increased notably due to its presence in high-profile malware families including Stuxnet [12], Conficker [36], and Flame [38]. Myriad proposals have appeared in the literature to protect against using USB storage devices for exfiltration and installing malicious payloads [11, 30, 37, 40]. Schumilo et al. present a USB fuzzer that can be used to harden drivers against exploitable software flaws, which improves security at the driver layer by improving the internal logic of device-specific drivers [35]. These security mechanisms operate at higher layers of the USB stack (i.e., the specific driver layer), which is insufficient to defend against BadUSB attacks.

Very few existing security solutions are positioned to defend against malicious USB firmware. Yang et al. propose a trust management scheme that mediates use of USB storage devices in industrial control systems [40]. While they consider the BadUSB attack, they conclude that they cannot prevent malicious storage device from requesting additional interfaces. Secure USB devices such as IronKey [18] can prevent BadUSB attacks by using signed firmware, provided that the device manufacturer is trusted and the signing key is kept safe. Unfortunately, these devices are costly, and have not overtaken traditional USB devices in most enterprise environments. Moreover, some organizations may be concerned about state-sponsored attacks in which the device manufacturer has been coerced into sharing their signing keys. GoodUSB provides additional assurance that even “Secure” USB devices are behaving correctly.

GoodUSB leverages virtual honeypots, which have also appeared elsewhere in the literature. Provos’ Honeyd system provides insight into network attacks by deploying virtual machines honeypots in arbitrary routing topologies [33]. Poeplau and Gassen present *Ghost*, a honeypot for USB storage [32]. The aim of Ghost is to detect the *propagation* of malicious USB storage payloads, which it accomplishes through emulating a storage device that periodically connects to potentially infected machines. In contrast, GoodUSB’s honeypot emulates a USB host, using hardware virtualization to hoist the entire USB controller into the virtual machine. Because it emulates the device and not the host, *Ghost cannot detect BadUSB attacks*. The GoodUSB honeypot provides a more general architecture that can profile malicious USB storage payloads as well as BadUSB attacks that covertly request privileged interfaces; this comes at an increased computational resource cost, as an entire host needs to be emulated instead of just a USB device.

A vital component of the GoodUSB architecture is the ability to identify devices that have previously connected to the host. Device identification is an especially difficult problem due to the unavailability of trusted hardware to bootstrap host-to-device attestation. As an alternative to trusted hardware, device fingerprinting schemes attempt to leverage innate characteristics (e.g., power, timing) of the device in order to establish identity. Fingerprint schemes for 802.11 devices have been proposed that leverage passive [13] and active [23] timing analysis, as well as radio frequency metrics [24, 4]. Gerdes et al. identify Ethernet cards through analysis of the analog signal of network packets [14]. Gupta et al. fingerprint electronic devices using the electromagnetic interference generated by switch mode power supplies [15]. Daneve et al. fingerprint RFID cards through extraction of the modulation shape and spectral features of the signals emitted by the transponder [9]. While we considered incorporating a fingerprint mechanism into GoodUSB, it is unclear if these approachs could be successfuly applied to USB Devices. Power analysis may be frustrated by the fact that the device is powered by the host. These fingerprint schemes also make the assumption of a benign target that is not attempting to evade detection; in our threat model, the adversary could attempt to modify its fingerprint to avoid detection, thereby complicating timing analysis. As a result of these pitfalls, GoodUSB does not attempt to perform device fingerprinting. Instead, it attempts to catch a malicious device in its lie by checking its requested interfaces against the user’s expectations.

Unlike host-to-device identification, a variety of proposals leverage the USB interface to perform device-to-host identification. Wang et al. [39] and Davis [10] independently observe that variations in protocol implementations leak information about the host operating system; however, these schemes would not be effective against an active fingerprint target, as demonstrated by Bates et al. [2]. Timing analysis of USB packets has also proven to be an effective means of identifying the host operating system [22], and even offers limited ability to differentiate between instances of identically-deployed machines [2]. Butler et al. present a mechanism for host verification that performs TPM attestations over the USB interface [6]; however, trusted hardware is a not a panacea due to the threat of cuckoo attacks [29].

GoodUSB’s user notifications contain a security image component. Graphical password systems were recently surveyed by Biddle et al. [3]. Passfaces is a recognition-based system for general authentication in which, during login, users choose a pre-selected face from a panel of candidate faces [5]. The Story system is also recognition-based; users select a portfolio of random images, and during login they must select their images in the correct sequence from amidst a portfolio of decoys. To prevent a BadUSB device from masquerading as another device, GoodUSB uses a recognition-based system to succinctly represent the claimed device identity. Unlike full-fledged graphical password systems, GoodUSB’s security images serve to provide an intuitive binding between a device and its requested interfaces. As a result, password guessing attacks are not a concern. Unfortunately, security images have been shown to be of limited utility on banking websites [34]. The strength of our security image component relies on the assumption that employees in an enterprise environment that have undergone security training will be more capable of identifying suspicious activity compared to typical banking customers.

## 8. CONCLUSION

USB attacks are becoming more sophisticated, affecting all classes of USB device instead of just storage. To date, there has been no practical defensive solution against BadUSB attacks, which expose the fundamental vulnerabilities of unconstrained privileges in USB devices. In this work, we present the design and implementation of *GoodUSB*, which enforces permissions of devices by encoding user expectations into USB driver loading. GoodUSB provides a security image component for better user experience and a honeypot mechanism for profiling suspicious USB devices. Outside of delays associated with user input, GoodUSB’s performance overhead during USB enumeration is just 5.2% (about 7 milliseconds). With this new method of constraining privilege of USB devices, users and administrators now possess a powerful new tool for securing their computers, permitting the re-introduction of these valuable devices back into the enterprise. The code and data used for GoodUSB as well as modified Linux distributions are available at our website, www.florida-security.org.

## 9. ACKNOWLEDGEMENTS

This work is supported in part by the US National Science Foundation under grant numbers CNS-1540217 and CNS-1540218, as well as by the Florida Cyber Consortium.

## 10. REFERENCES

[1] M. Al-Zarouni. The Reality of Risks from Consented Use of USB Devices. *School of Computer and Information Science, Edith Cowan University, Perth, Western Australia*, 2006.

[2] A. Bates, R. Leonard, H. Pruse, K. R. Butler, and D. Lowd. Leveraging USB to Establish Host Identity Using Commodity Devices. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS '14, February 2014.

[3] R. Biddle, S. Chiasson, and P. Van Oorschot. Graphical Passwords: Learning from the First Twelve Years. *ACM Comput. Surv.*, 44(4):19:1–19:41, Sept. 2012.

[4] V. Brik, S. Banerjee, M. Gruteser, and S. Oh. Wireless Device Identification with Radiometric Signatures. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, MobiCom '08, Sept. 2008.

[5] S. Brostoff and M. Sasse. Are Passfaces More Usable Than Passwords? A Field Trial Investigation. In *People and Computers XIV – Usability or Else!*, pages 405–424. Springer London, 2000.

[6] K. Butler, S. McLaughlin, and P. McDaniel. Kells: A Protection Framework for Portable Data. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, Austin, TX, USA, Dec. 2010.

[7] A. Caudill and B. Wilson. Phison 2251-03 (2303) Custom Firmware & Existing Firmware Patches (BadUSB). *GitHub*, 26, Sept. 2014.

[8] Compaq, Hewlett-Packard, Intel, Microsoft, NEC, and Phillips. Universal Serial Bus Specification, Revision 2.0, April 2000.

[9] B. Danev, T. S. Heydt-Benjamin, and S. Capkun. Physical-layer Identification of RFID Devices. In *Proceedings of the 18th USENIX Security Symposium*, Aug. 2009.

[10] A. Davis. Revealing Embedded Fingerprints: Deriving Intelligence from USB Stack Interactions. In *Blackhat USA*, July 2013.

[11] S. A. Diwan, S. Perumal, and A. J. Fatah. Complete security package for USB thumb drive. *Computer Engineering and Intelligent Systems*, 5(8):30–37, 2014.

[12] N. Falliere, L. O. Murchu, and E. Chien. W32. Stuxnet Dossier. 2011.

[13] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker. Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting. In *Proceedings of the 15th USENIX Security Symposium*, Aug. 2006.

[14] R. M. Gerdes, T. E. Daniels, M. Mina, and S. F. Russell. Device Identification via Analog Signal Fingerprinting: A Matched Filter Approach. In *Proceedings of the 2006 Network and Distributed System Security Symposium*, NDSS '06, Feb. 2006.

[15] S. Gupta, M. S. Reynolds, and S. N. Patel. ElectriSense: Single-point Sensing Using EMI for Electrical Event Detection and Classification in the Home. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, UBICOMP '10, Sept. 2010.

[16] Hak5. Episode 709: USB Rubber Ducky Part 1. http://hak5.org/episodes/episode-709, 2013.

[17] Hak5. USB Rubber Ducky Payloads. https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads, 2013.

[18] Imation. Ironkey. http://www.ironkey.com/en-US/resources/, 2013.

[19] Imation. IronKey Secure USB Devices Protect Against BadUSB Malware. http://www.ironkey.com/en-US/solutions/protect-against-badusb.html, 2014.

[20] S. Kamkar. USBdriveby. http://samy.pl/usbdriveby/, 2014.

[21] J. Lee, L. Bauer, and M. Mazurek. The effectiveness of security images in internet banking. *Internet Computing, IEEE*, 19(1):54–62, Jan 2015.

[22] L. Letaw, J. Pletcher, and K. Butler. Host Identification via USB Fingerprinting. In *IEEE Sixth International Workshop on Systematic Approaches to Digital Forensic Engineering*, SADFE '11, May 2011.

[23] D. Loh, C. Y. Cho, C. P. Tan, and R. S. Lee. Identifying Unique Devices Through Wireless Fingerprinting. In *Proceedings of the 1st ACM Conference on Wireless Network Security*, WiSec '08, Apr. 2008.

[24] N. T. Nguyen, G. Zheng, Z. Han, and R. Zheng. Device Fingerprinting to Enhance Wireless Security Using Nonparametric Bayesian Method. In *Proceedings of the 30th IEEE International Conference on Computer Communications*, INFOCOM '11, Apr. 2011.

[25] K. Nohl and J. Lehl. BadUSB – On Accessories That Turn Evil. In *Blackhat USA*, Aug. 2014.

[26] OLEA Kiosks, Inc. Malware Scrubbing Cyber Security Kiosk. http://www.olea.com/product/cyber-security-kiosk/, 2015.

[27] Open Source Security,Inc. grsecurity. https://grsecurity.net/, 2013.

[28] OPSWAT. Metascan. https://www.opswat.com/products/metascan, 2013.

[29] B. Parno. Bootstrapping Trust in a "Trusted" Platform. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security*, HotSec '08, Aug. 2008.

[30] D. V. Pham, M. N. Halgamuge, A. Syed, and P. Mendis. Optimizing Windows Security Features to Block Malware and Hack Tools on USB Storage Devices. In *Progress in Electromagnetics Research Symposium*, 2010.

[31] PJRC. Teensy 3.1. https://www.pjrc.com/teensy/teensy31.html, 2013.

[32] S. Poeplau and J. Gassen. A Honeypot for Arbitrary Malware on USB Storage Devices. In *7th International Conference on Risk and Security of Internet and Systems*, CRiSIS '12, Oct. 2012.

[33] N. Provos. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[34] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *28th IEE Symposium on Security and Privacy*, SP'07, May 2007.

[35] S. Schumilo, R. Spenneberg, and H. Schwartke. Don't trust your USB! How to find bugs in USB device drivers. In *Blackhat Europe*, Oct. 2014.

[36] S. Shin and G. Gu. Conficker and Beyond: A Large-scale Empirical Study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 151–160, New York, NY, USA, 2010. ACM.

[37] A. Tetmeyer and H. Saiedian. Security Threats and Mitigating Risk for USB Devices. *Technology and Society Magazine, IEEE*, 29(4):44–49, winter 2010.

[38] J. Walter. "Flame Attacks": Briefing and Indicators of Compromise. *McAfee Labs Report*, May 2012.

[39] Z. Wang and A. Stavrou. Exploiting Smart-phone USB Connectivity for Fun and Profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[40] B. Yang, D. Feng, Y. Qin, Y. Zhang, and W. Wang. TMSUI: A Trust Management Scheme of USB Storage Devices for Industrial Control Systems. Cryptology ePrint Archive, Report 2015/022, 2015. http://eprint.iacr.org/.

[41] yubico. yubikey. https://www.yubico.com/products/yubikey-hardware/, 2015.

[42] P. Zaitcev. The usbmon: USB Monitoring Framework. http://people.redhat.com/zaitcev/linux/OLS05_zaitcev.pdf, 2005.