USBlock: Blocking USB-based Keypress Injection Attacks

Sebastian Neuner[1], Artemios G. Voyiatzis[1], Spiros Fotopoulos[2], Collin Mulliner[3], and Edgar R. Weippl[1]

[1] SBA Research, Austria
{sneuner, avoyiatzis, eweippl}@sba-research.org
[2] University of Patras, Greece
spiros@physics.upatras.gr
[3] mulliner.org, USA
collin@mulliner.org

Abstract. The Universal Serial Bus (USB) is becoming a prevalent attack vector. *Rubber Ducky* and *BadUSB* are two recent classes of a whole spectrum of attacks carried out using fully-automated keypress injections through innocent-looking USB devices. So far, defense mechanisms are insufficient and rely on user participation in the trust decision.
We propose USBlock, a novel approach to detect suspicious USB devices by analyzing the *temporal* characteristics of the USB packet traffic they generate, similarly to intrusion detection approaches in networked systems.
Our approach is unique in that it does not to involve at all the user in the trust decision. We describe a proof-of-concept implementation for Linux and we assess the effectiveness and efficiency of our approach to cope with temporal variations in typing habits and dynamics of legitimate users.

Keywords: Security, USB, BadUSB, Linux kernel, system security

1 Introduction

The Universal Serial Bus (USB) is by far the most widely-used connector for modern computer systems. It is used to connect a plethora of peripheral devices to computers, including keyboards, mice, cameras, printers, and storage media. Many different attack vectors abuse the pervasiveness of USB, as for example dropping USB thumb drives on parking lots for users to pick up and attach on their computers [19]. As network-based defenses steadily improve and can block efficiently the malicious network traffic reaching an organization, USB becomes an attractive entry point for penetrating an organization.

Under the hood, USB is more than a simple connector. It is a complex *communication protocol*, often implemented and offered as a firmware. Lately, there are devices on the market with the ability to update their USB firmware. This

capability has been exploited as a subtle attack vector, hiding malicious functionality on an abstraction layer that modern computer antivirus cannot cope with. BadUSB[4] and Rubber Ducky[5] classes of attacks are successful demonstrations of the attack feasibility. The associated threat is rather high: at this level the device interfaces directly with device drivers that run in the most privileged level of modern consumer-grade operating systems (e.g., as ring-0 modules of the Linux kernel).

In this paper, we provide insights on how these USB-protocol-level attacks work and explore how their attack patterns can be detected at the system level. Such an approach relieves the end user from being involved in trust decisions and thus, makes the security of the system less dependent to user actions. Specifically, we make the following contributions:

- We study the behavioral characteristics of Rubber Ducky and BadUSB classes of attacks.
- We devise criteria for automating their detection.
- We design, implement, and evaluate a simple yet *very effective and extensible* system-level countermeasure based on USB packet traffic analysis to detect and defend against such attacks *without* requiring user intervention.

The rest of the paper is structured as follows: Section 2 provides the necessary background information for the USB protocol and related attacks and defenses. Section 3 studies the temporal characteristics of USB-based attacks and proposes a system-level defense mechanism. Section 4 describes a prototype implementation of our proposal and evaluates its efficiency in real-world settings. Finally, Section 5 presents our conclusions and discusses future directions of work.

2 Background

2.1 The USB protocol

USB is the most widely-used computer peripheral connector today. USB 3.2 is its latest revision. The USB device communication is based on a tiered-star topology with one dedicated master controller. Besides the controller, a hub manages the connected USB devices. If the master controller acts as a hub too, then the hub is called the "root hub". Every USB hub uses seven bits to address connected USB devices. This leads to a limit of 127 attachable USB devices per hub.

The connection of a USB device to a hub works as follows[6]: The USB hub waits for new devices to be plugged in. Upon connection, channels for communication are created: the so-called "endpoints", acting as sources and sinks of data. The endpoints are logically grouped together to "interfaces" and are announced to the host via "interface descriptors".

[4] `https://srlabs.de/badusb/`
[5] `https://github.com/hak5darren/USB-Rubber-Ducky`
[6] `http://www.beyondlogic.org/usbnutshell/usb3.shtml`

The USB communication is realized by exchanging "USB packets" over the shared serial bus. The USB protocol defines four transfer types (Control, Isochronous, Interrupt, and Bulk) and three packet types (Token, Data, and Status).

Modern operating systems, including Microsoft Windows, Linux, and Apple Mac OS, utilize the information collected by the hub from the connected USB devices to (dynamically) load the appropriate device drivers. For each announced interface descriptor of a device, the operating system combines the device-provided device class, interface class, and vendor and product identifiers (VID and PID respectively) to decide which capabilities are provided by the device and to bind the appropriate device driver(s).

As an example, a modern USB mouse may offer the capabilities of a human interface device (HID) and those of a display (e.g., to display its sensitivity level). Or a USB headset may offer the capabilities of an audio output device and that of a constrained HID for its volume up/down and mute buttons. In such cases, the devices will have two different interfaces and each of them will get a different driver bound to it.

2.2 USB protocol security

The USB protocol does not dictate a form of device authentication. Rather, every USB hub *blindly trusts* any information announced by the connected device about their capabilities. We note that modern USB devices incorporate, for legitimate reasons, multiple functionalities (e.g., a mouse announcing itself also as a display device). Such functionalities are hard for a user to link together and reason for any associated risk. These combined with the wide prevalence of USB devices, render USB an attractive attack vector [1,21].

In the past, the entry barrier for realizing attacks based on the inherent weaknesses of the USB protocol was very high. It dropped significantly by the time USB firmware chips with reflashing capabilities became available on the open market[7]. On the one hand, firmware updates for consumer products are often a necessity due to shortened time-to-market and insufficient testing. The alternative would be a product recall which would cause a logistics nightmare. On the other hand, firmware updates significantly lower the resources and expertise for launching USB-based attacks.

Figure 1 depicts the principle of a USB device and endpoint setups, which occur during the Control transfer phase using Setup packets. In this example, the device announces support for two functionalities, namely a mass storage and a keyboard (HID device). The former seems like a normal behavior, assuming that the user plugged in a USB thumb drive or external disk. In this case, the user expects that the operating system (host) will load the mass storage device driver and be able to further interact with the storage device. However, we note that without having knowledge of the device specifics, this announcement could also have been an attack vector.

[7] `https://adamcaudill.com/2014/10/02/making-badusb-work-for-you-derbycon/`

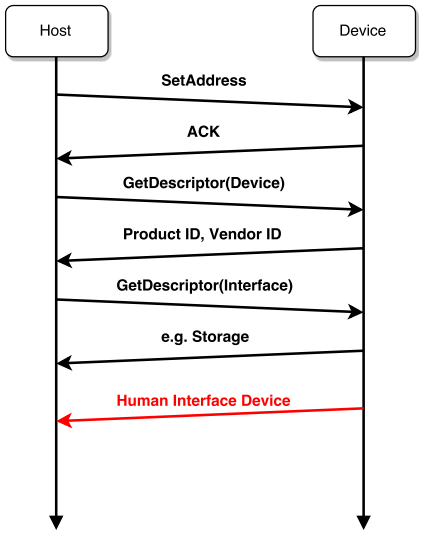


Fig. 1: USB packet sequence diagram (malicious behavior)

The latter announcement forces the host to bind a keyboard driver as well. If the announcement comes from a modified, malicious firmware, then the first step of the attack is already successful. The firmware then launches the second step of its attack. This involves sending keypresses from the (non-existent) keyboard. This "USB-based keypress injection attack" assumes that the system interaction caused by these keypresses will go unnoticed by the user and, thus, will succeed to deliver the malicious actions (e.g., download from the Internet or access from the USB storage and then execute a zero-day malware).

So far, various attacks exploiting the inherent weaknesses of the USB protocol have been proposed [13]. We review the most characteristic of them in the following paragraphs.

2.3 BadUSB and IRON-HID

The BadUSB attack enables a USB storage device to act not only as a SCSI device (mass storage), but also as an HID one. By acting as a keyboard, the data coming from the connected USB device is interpreted as keypresses. An attack can install a backdoor to the host system or "call home" over the network for example. From that point on, the attacker has total control over the infected host system.

Researchers and practitioners work both on improving BadUSB-like attacks and reducing their attack surfaces. In the so-called "IRON-HID" attack, additional programmable hardware (e.g., a Teensy board[8]) is hidden in places like

[8] `https://www.pjrc.com/teensy/`

keyboards and portable USB batteries [7]. By connecting a smartphone to the crafted USB battery via a crafted USB On-The-Go cable, the smartphone is switched into USB host mode. From that moment on, the smartphone is able to eavesdrop on all USB communications. IRON-HID can be used also to inject fake keypresses with the aim to brute-force the Android screen unlock PIN.

2.4 Rubber Ducky

The *Rubber Ducky* is a physical device designed by the Hak5 group[9]. The Rubber Ducky works as a normal keyboard when it comes to driver binding: it simply needs the operating system's HID driver to work. However, the Rubber Ducky delivers USB-based payloads (keypresses pre-defined by an attacker) upon being connected to the victim system.

The pre-defined keypresses are written in Ducky Script, a simple-to-use scripting language[10]. Once the payload is developed, it is compiled into a binary and placed on the microSD card of the Rubber Ducky device. Upon connecting the Rubber Ducky to a host computer, the built-in Atmel AT32UC3B1256 chip of Rubber Ducky emulates the pre-defined keypresses in the *fastest* rate the USB port can deliver and the device driver of the attacked system can handle.

2.5 BadAndroid

BadAndroid[11], like BadUSB, adds malicious functionality to an otherwise benign Android device. In contrast with BadUSB, the firmware of the Android device needs not be flashed.

A possible attack scenario using BadAndroid looks like the following: using social engineering, an attacker pretends that she needs to charge the battery of her Android smartphone and asks to plug it in to the target's laptop. While the smartphone is connected for charging, BadAndroid actually alters the routing table of the host (laptop) system without the user noticing, i.e., it changes the default network gateway of the laptop to be the IP address of the Android smartphone. From that moment on, all the network traffic of the laptop is routed via the smartphone, enabling the attacker to inspect and alter the whole bi-directional network traffic.

In a second attack scenario, BadAndroid could change the entries for the laptop's DNS servers and, therefore, redirect the laptop's traffic to servers controlled by the attacker.

2.6 BadBIOS

A maliciously-crafted BIOS hidden on the USB device could be installed on the computer by emulating keypresses at boot time[12]. This BadBIOS overrides

[9] `https://hak5.org`

[10] `https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Duckyscript`

[11] `https://srlabs.de/blog/wp-content/uploads/2014/07/BadAndroid-v0.1.zip`

[12] `https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf`

the original BIOS and becomes the default BIOS to boot from. This allows an attacker to execute commands even before the actual operating system is loaded.

The applicability of BadBIOS is demonstrated[13] for modern Smart TVs. In this case, a Smart TV is forced to produce high-frequency audio signals. These signals contain information which is then transmitted to other devices.

2.7 Other attacks

The published literature includes USB-based attacks that exploit weaknesses beyond the ones spawned by BadUSB. The *teensy* USB development board[14] is designed to instrument and audit USB drivers, ports, and related software. This board is reprogrammable and can be used to launch attacks that are based on emulating keypresses and mouse movements.

The current version of the *USBdriveby* attack tool[15] targets Apple iOS devices. Once successful, USBdriveby alters the routing entries of the attacked system so as to redirect traffic to spoofed websites. The tool also installs a backdoor for the case the user detects the route modifications and changes them back to the legitimate ones.

2.8 Defenses against USB-based attacks

There have been proposals in the literature for defending against USB-based attacks, especially after the Stuxnet malware, which spread via infected USB drives and penetrated air-gapped systems [10,6]. A first attempt towards more organizational security of USB device security is described in [22]. There, a trust management scheme, namely TMSUI, is proposed. TMSUI protects an ICS by allowing the connection of USB storage devices *only* on certain protected terminals and *only* for a specific amount of time.

ProvUSB is an architecture for fine-grained provenance collection and tracking on smart USB devices [17]. ProvUSB aims at environments where the use of pre-approved-only USB drives can be controlled and enforced.

UScramBle is a proposal for protecting against eavesdropping attacks that are feasible due to the broadcast nature of (pre-USB 3.0) hub-to-device communication [12]. It can be used to defend against reverse engineering of legitimate devices.

A line of defense against BadUSB-like attacks incorporates the user into the trust decision. One example is USBWall [9]. USBWall uses a Beagle Board[16] in order to enumerate USB devices on behalf of the operating system. As soon as the enumeration is carried out, the user is asked to decide whether the USB device must be removed from the system or is safe for further use. USBCheckIn

[13] `https://nakedsecurity.sophos.com/2015/11/16/badbios-is-back-this-time-on-your-tv/`
[14] `https://www.pjrc.com/teensy/`
[15] `http://samy.pl/usbdriveby/`
[16] `https://beagleboard.org`

is a hardware-based approach, where the user is forced to actively interact with the HID using guided patterns so as to authorize its use [5].

The "G DATA USB Keyboard Guard" software[17] is another system that relies on the users' decision whether a USB device is malicious or benign. There, when a new HID device is connected to the protected computer, the software asks the user to decide if the device interface(s) are to be trusted or not. Once a decision is made, the device (in fact, the combination of product and vendor identifiers) is either whitelisted or blacklisted so as to avoid asking again in the future. If an attacker flashes a malicious device to present with a previously whitelisted combination of product and vendor identifiers (e.g., a legitimate keyboard), then their attack will go unnoticed.

GoodUSB is a similar approach described in [16]. GoodUSB includes a Linux kernel module that maps USB devices to specific whitelisted drivers. Upon connection of a new USB device, GoodUSB involves the user to decide if it should allow or deny the new device. If the user marks the device as malicious, the control is transferred to a virtualized USB honeypot running on QEMU-KVM. This allows to monitor and profile the activity of the USB device for further analysis.

`USBFILTER` is a packet-level filter (firewall) for USB communications developed for the Linux kernel space [18]. The user defines access rules in `USBTables`, the userland component of `USBFILTER` and the kernel-space component checks each USB packet received for match with one of the rules and decides to either forward or drop it. By design, `USBFILTER` supports only per-packet processing. Given the simplicity of the supported rules, attackers can evade the rules by adjusting their behavior accordingly; overall `USBFILTER` is a *deterministic* solution that detects already known attacks and does not have any anomaly detection capabilities [13].

Cinch is an approach similar in principles to `USBFILTER` [2]. However, Cinch isolates all USB devices from the host and passes communication through a virtual machine acting as a gateway that enforces the access policies.

SandUSB offers a GUI for the users to mark a newly plugged-in device as malicious or benign [11]. Should the users consider the device being malicious, they can either blacklist it or redirect it to a USB sandbox for further analysis.

3 A Novel Approach for Detection of USB-based Keypress Injection Attacks

We consider a keypress injection attack as the most severe USB-based threat to system security. This is because a carefully-crafted attack, launched through innocent-looking keypresses, leverages the powerful resources and flexibility of the host system so as to take over the full control of the system itself.

The proposed defenses against keypress injection attacks until now have in common that they rely at some point on user decisions. The user insights in such

[17] `https://www.gdatasoftware.com/en-usb-keyboard-guard`

low-level system trust decisions is not an optimal solution. This is especially true when such interactions break their mental model for the primary task at hand, so as to cope with a secondary one [3,4].

In the following, we explore how we can detect and defend against USB-based keypress injection attacks by performing USB packet traffic analysis at the system level, *without* involving the user in the decision loop. Our aim is to simplify attack detection and offer neutralization upon connection of a malicious USB device that acts as a keyboard. This includes fast detection and no user involvement in the security decision.

3.1 Threat model

We assume an enterprise environment where computers are equipped with USB ports and the users are free to plug in and unplug USB devices for their day-to-day work duties (e.g., mass storage devices, headsets, and web cameras for teleconferences).

We further assume that the attackers succeed in connecting a crafted device with a malicious USB firmware to one or more of these computers. This can be achieved by the attackers themselves, if they have physical access to the targeted computer, inside or outside the premises of the enterprise. Or, by handing in a malicious device to legitimate users and exploit their curiosity (e.g., drop a USB thumb drive in their postal mailbox) or apply a social engineering attack vector (e.g., "can you please print the file from my USB thumb drive?").

We do not consider attack vectors such as USB storage media loaded with malware (e.g., exploit the "autorun" feature). Mitigations for such attacks are already offered by commercial antivirus products [14]. We also do not consider attacks that exploit (unknown) vulnerabilities of the USB device drivers of the host operating system triggered by malformed USB packets [8]. Rather, we assume that all USB packets are well-formed and valid according to the USB protocol.

3.2 Patterns of keypress injection attacks

The first step for developing appropriate defenses is to get a better understanding of the keypress injection attack patterns. Towards this direction, we experimented with two USB device types that their firmware can be updated and for which appropriate reflashing tools are available. The first one was a Rubber Ducky device. The second one was a Toshiba USB 3.0 USB drive.

In the case of Rubber Ducky, it sufficed to compile a new firmware, which contained an attack payload, by using the provided compiler and then copy the firmware on its SD card.

The case of the benign Toshiba USB drive required some additional steps. First, the original firmware of the device was dumped, then the attack payload was integrated in the firmware, and finally the firmware was written back to the chip. This was not an error-free process. Our first attempts ended up with

unusable (bricked) devices and unreliable functionality of the controller chip resulting in unstable behavior.

We also prepared a desktop computer that acted as the host for the attacks. We used the Wireshark network protocol analyzer[18] to monitor the USB connections and collect the related USB packet traces for further analysis.

As an attack demonstration scenario, we opted to use the automatic launch of a text editor in the Linux operating system, including some text filling. Once connected, the malicious devices registered themselves as keyboards and sent the necessary keypresses. The sequence of events was as follows:

1. An artificial delay of 500 milliseconds.
2. Send the `ALT` key followed by the `F2` key in order to prepare an application launch.
3. An artificial delay of 500 milliseconds.
4. Send a string of characters for launching a text editor (e.g., `gEdit` or `mousepad`), followed by the `ENTER` key.
5. An artificial delay of 500 milliseconds.
6. Send one paragraph of text comprising 515 characters from the Bacon Ipsum (`http://baconipsum.com/`) text.

Albeit not malicious in nature, the attach scenario above serves as a baseline for building weaponized attacks, such as opening a terminal, disabling any running antivirus service, and running a `wget` command to download malicious code to the attacked system. The artificial delays are necessary to provide the operating system with enough time to successfully respond to issued commands, e.g., opening the text editor. The attack script sends 526 keypresses in total.

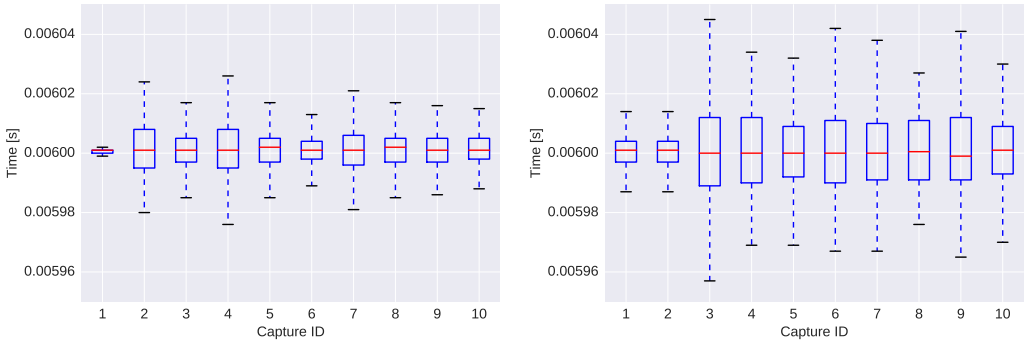
We ran each attack (BadUSB and Rubber Ducky) ten times and collected in total 20 Wireshark traces. We analyzed these traces and focused on the timing patterns of the `KEY_DOWN` events that are sent when a key on a keyboard is pressed.

Figure 2 depicts the distribution of the distance between each of the 5,260 recorded events (i.e., the interarrival time of the keypresses) for each of the ten repetitions of each attack in box-and-whisker diagrams. There are only a couple of outliers for each trace (Capture ID), which are at about 1.35 seconds (in the case of BadUSB) and 1.00 seconds (in the case of Rubber Ducky). The median value in both cases is about 6 ms and almost all values are concentrated in a narrow band around this value, as depicted in Figure 2a and Figure 2b.

This was a stable behavior in all traces and for both attacks; Rubber Ducky exhibits a greater variability, but still within the narrow band. We ran repeated experiments with all the payloads made available by the Rubber Ducky authors[19]. There were in total more than 70 different payloads at the time of writing. The analysis of the traces revealed that *all* of the payloads produced the keypresses with no delay, i.e., the USB-based keypress injection attacks try to conclude their malicious actions as fast as possible.

[18] `https://wireshark.org/`

[19] `https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads`



(a) BadUSB timings

(b) Rubber Ducky timings

Fig. 2: Interarrival times of `KEY_DOWN` events in collected traces (outliers exclused for the sake of clarity)

From an attack point of view, it is a rational choice to inject keypresses as fast as possible: for an attack to be successful, all the events of the script must execute without being interrupted by any user-initiated typing activity. Since the attacked system has two keyboards now, it is possible that the user continues their typing activity. In this case, typing will intermix with the (fake) keypresses of the attack script and, thus, neutralize the latter by accident (and possibly frustrate the user with the extra characters). The less the time in between the (fake) keypresses, the more the probability of a successfully-launched attack. This is a key observation for the design of our defense.

3.3 Keystroke dynamics

Research in *keystroke dynamics* (or *keystroke-based biometrics*) suggests that human typing patterns exhibit variations and these "typing dynamics" can be strong enough to be used for authentication or identification purposes [15]. Also, human beings cannot perform better than 80 ms between their keypressess [20]. In contrast, our analysis reveals that BadUSB-like attacks inject keypresses at an almost stable rate of 6 ms. These two rates differ by an order of magnitude.

3.4 USBlock: blocking malicious USB packet traffic

USBlock is a defense we devised that exploits the temporal gap between human and BadUSB-like attack typing dynamics. The design assumption is that a USB host monitor (i.e., USBlock) has access to precise timing information of received USB packets and is able to distinguish *fast* between the normal typing behavior of human beings and the abnormal keypress sequence (AKS) timings of BadUSB-like attacks.

A simple case of an AKS is a "*rapid (keypress) event sequence*" (RES). We define a RES using two components, a time threshold value t and a sequence threshold value s. We say that a RES occurred whenever we observe a sequence of s consecutive keypresses with an interarrival time less than t seconds between each of them. If a RES occurs, a defense action must be taken.

The selection of the exact values for t and s is a design choice. The t-threshold can be any value $0.006 < t < 0.08$, i.e., anything between the 6 ms median value of the BadUSB-like attacks and the lower limit of 0.08 seconds for humans. A t close to the lower bound of 0.006 makes USBlock *more prone* to false negatives, thus, risking a successful attack going unnoticed. Yet, it reduces the probability of false positives, as humans cannot type that fast. In contrast, a t close to the upper bound 0.08 makes USBlock *more prone* to false positives, thus, risking user complaints on legitimate usage scenarios. Our analysis of the collected attack and human-typing traces (cf. Section 4) suggests that a $t = 0.02$ is a sufficient threshold for the current generation of BadUSB-like attacks.

An $s = 1$ makes USBlock *blindly aggressive* (many false positives), penalizing even in a single occurrence of a keypress under the timing threshold. In contrast, a large s (e.g., over 100) allows USBlock to make *confident decisions* on the presence of an AKS. However, one must account two additional points. First, if the decision algorithm does not react in real-time, it risks to allow an attack to have occurred already by the time a (correct) decision is made. This is unacceptable from a security point of view. Second, the length of the attack vector, i.e., the number of keypresses injected to realize the attack, might be lower than s (e.g., only 20 or 30 keypresses compared to an $s = 100$). In this case, USBlock will fail to detect the attack altogether, as there are not enough "malicious" keypresses present to react on.

Our analysis of the available traces suggests that a short $s = 3$ offers the clear advantage that a keypress injection attack is detected and prevented just after the very first few fast keypresses are sent. The attack traces *contained not a single case* where two or more outlier values came in pairs or bursts. Furthermore, to the best of our knowledge, no attack vector can be realized with only three keypresses. Hence, $s = 3$ is an excellent choice.

The capabilities of USBlock are better demonstrated with an example of a defense realization against current generation of Ducky Script malicious payloads. A USB monitor runs as a piece of software on the host system and measures the interarrival times of keypresses sent by each connected USB device for a RES, using the t and s parameters. When a RES is detected, the USB monitor instructs the USB hub to switch off the power of the USB port for some (e.g., ten) seconds (configurable). It also instructs the operating system to unload (unbound) the respective device driver. This in effect blocks access to the suspicious device. We highlight that *no additional* piece of hardware is needed for performing these actions. The disconnection approach ensures that no user involvement is required in restoring the connectivity of other USB devices that are possibly attached to the specific USB port where the attack occurred.

One may argue that the design opts for a rather aggressive reaction to suspicious events. We believe that our two-step disablement approach accommodates this. It is better for users to notice an occasional interruption of their normal flow (in case the events are indeed produced that fast by human beings) rather than risking an infection by a malicious device. If the interruptions become too frequent, it is an indication of an attack in progress (e.g., a USB drive trying to relaunch the attack). In this case, it would be better for the IT support personnel to inspect the offending system.

3.5 USBlock defenses against advanced attacks

USBlock with RES defends currently against *all known* malicious Ducky Script payloads. The approach of USBlock is generic enough and allows to realize and integrate new defenses against new attacks. As the cat-and-mouse game between attackers and defenders might evolve for BadUSB-like attacks, RES can be replaced by a more complex AKS detection logic. Such logic will push attackers to adjust keypress injection rates to mimic human behavior. This task will be more and more difficult to both realize in the constrained environment of USB firmware and to match specific user typing patterns. At the moment, this is not deemed necessary and would incur additional and unnecessary processing overhead.

We note that USBlock with RES cannot be defeated by malicious firmware that delays the launch of the payload (i.e., the start of the attack), once the fake device is plugged. USBlock is continuously monitoring for keypress events. Hence, when a RES occurs, USBlock will detect it, no matter how long ago the fake device was plugged.

One may argue that a malicious Ducky Script can introduce delays between keypresses to avoid detection. However, the constant interarrival time of the keypresses is an easily-detectable AKS pattern. Should a Ducky Script can generate random delays between keypresses, these must occur as a human typing pattern, sparse in time. At this rate, they will become intermixed with the normal typing activity of the user (and thus, neutralized) or they will become noticeable by the user, as the attacks do not happen "in the blink of an eye" anymore (e.g., the cursor is moved or the focus of the working window is lost). Should the user is incapable or unwilling of noticing the additional typing activity on their monitor, we must resort on enterprise network security defenses that might be able to detect the malicious activity of the attacked system, once infected.

Clearly, analyzing the typed *command strings* (from humans or scripts) and reasoning about their (possibly) malicious *intentions* is beyond the scope of USBlock. Such fine-grained keypress analysis will probably not be acceptable by the users, as it severely violates their privacy creating an Orwellian feeling of constant monitoring.

One may argue that a malicious USB firmware can monitor the status of the operating system and launch the payload during periods of user inactivity (e.g., when there is no typing activity for a long time). We are not aware of such capabilities for Ducky Scripts and, to the best of our knowledge, such information

cannot be requested by the USB firmware from the operating system over USB packets. Even if such capabilities exist in first place, USBlock is on the side of the operating system. Thus, it can also access such information and integrate them into its decision logic to block malicious USB traffic. We further note that such an attack cannot be launched at all if the screen is locked (e.g., due to inactivity or a precautious measure by the user when plugging an untrusted device).

3.6 Limitations of USBlock

While USBlock defends successfully against keypress injection attacks, it does not come free of limitations. USBlock lies and relies on information at the level of the operating system. As such, it cannot defend against attacks launched at the BIOS level, as is the case of BadBIOS discussed in Section 2.

Hardware keys, like Yubico YubiKey[20], are a popular means of two-factor authentication. Such devices identify themselves to the operating system as keyboards and "type" one-time passwords and other sensitive information on behalf of their owner. Hence, USBlock will interpret the YubiKey rapid keypresses as an attack. To overcome this, USBlock implements internally the following check for YubiKey devices: if a connected USB device reports a (VID, PID) of Yubico, the RES logic is disabled for this device. Rather, USBlock monitors the USB device traffic to ensure that the USB packet payload comprise exclusively "MODHEX" characters[21]. This is an improved approach over (VID, PID) whitelisting [16], as anyone can fake the reported (VID, PID). As discussed earlier in Section 2, these are inherent limitations of the USB protocol itself rather than of USBlock. The host system must blindly trust the information provided by peripheral devices without any authentication support.

4 Proof-of-concept implementation and evaluation

We now describe a proof-of-concept implementation of the designed system and its evaluation in realistic environments. The evaluation comprises two parts: the first part relates to the long-term stability of the prototype and to evaluating the effect of the temporal variations in typing habits; the second part explores the typing dynamics of different users.

4.1 Proof-of-concept implementation

We realized a proof-of-concept implementation for the Linux operating system comprising two parts[22]. The first part is a Linux loadable kernel module (for

[20] https://www.yubico.com/products/yubikey-hardware/

[21] The modified hexadecimal characters are {b, c, d, e, f, g, h, i, j, k, l, n, r, t, u, v}, cf. `https://www.yubico.com/support/knowledge-base/categories/articles/one-time-password-otp/`.

[22] We will release the code as open-source software hosted on GitHub once the double-blind review process concludes.

kernel version 4.2) that monitors for the keypresses. Being in the kernel, this part is as close as possible to the raw information about keypresses received from the USB device and enriches them with very precise timings. The kernel module then forwards the enriched information to the second part residing in userland. The communication between the two parts occurs over a `netlink` socket that is registered in both the kernel space and userland. This is an approach similar to the one of [18]. However, the latter does not support timestamps and multi-packet processing, which are necessary for our aims.

The second part is a Python script. This part implements the rapid event sequence (RES) detection logic and is responsible for unbinding the offending USB driver from the kernel. This effectively disconnects the device interface from the system. If the driver of the device is automatically re-bound (as part of an ongoing attack), then, as an additional protective measure, the driver for the corresponding USB hub on which the device is connected is also unbound for ten seconds. This effectively removes all the devices connected on this hub. The unbind/re-bind procedure is repeated until a user action is initiated or a system administrator takes over. We note that the USB packet processing, from kernel capture to RES detection (via the Python script) and reaction, requires on average about 0.3 ms per packet. Hence, it takes 1 ms to detect an $s = 3$ RES. This is more than enough processing time, given that the median interarrival time for BadUSB-like keypress injections is about 6 ms.

4.2 Evaluation of temporal variations

We evaluated the effect of temporal variations in typing as follows. We installed the prototype in one of the authors' computers, a notebook running Ubuntu 15.10. The notebook was used on a day-to-day basis with a USB-connected keyboard for a period of three months. The aim was twofold: on the one hand, to evaluate the stability of the prototype and to discover any problems that it might cause; on the other hand, to study the user's typing behavior in the course of a long period that involved a multitude of typing activities including code development, debugging, system administration tasks, scientific paper writing, preparation of talks and presentations, shell scripting, email typing, and web browsing.

We instrumented the kernel module `usbmon` [23] to collect and log the USB events and act as a middleware between the kernel and our offline analysis tools. In userland, the `TShark` part of the Wireshark bundle was used to "listen" to USB events sent by `usbmon`. A Python script was used to generate `pcap`-formatted files containing all the USB-related events. A second Python script was then used to process the files and store the collected information into a database for later analysis.

Overall, we did not experience any kind of stability problems while operating the prototype and the additional monitoring infrastructure. Neither we experienced any kind of measurable performance degradation. It did not cause any side effects (at least that we became aware of). Thus, the temporal variations in typing did not affect the operation of the prototype.

We collected in total timing information for more than 466,000 keypresses over more than 60 working days. Less than 1% of them were below the $t = 0.02$ second threshold, while the vast majority ranked quite higher.The median value of interarrival times was 0.10 seconds, while the average time was 0.21 seconds.

There was *no single case* of three or more consecutive keypresses with an interarrival time below the defined threshold. Thus, it was never the case that the prototype unbound the USB keyboard device driver, i.e., we experienced *no false positives*.

4.3 Evaluation of typing dynamics

The second phase of the evaluation was a study on the effect of the typing dynamics. We designed a small-scale user study so as to collect evidence about typing patterns and compare them against the behavior of the Rubber Ducky and BadUSB attacks as well as published literature for typing dynamics.

We developed a research prototype system comprising a headless Raspberry Pi Model 2B running Ubuntu server 15.10, a USB keyboard, and a battery pack for autonomous operation. Similarly to the previous evaluation, we used a kernel module to collect the keypress timing information and send them to a userland application. The latter collected and aggregated the information prior to storing them into a database for later processing.

We recruited 33 volunteers from our organization for this experiment. We visited each participant at their desk and asked them to type a short text in the comfort of their desk using our research prototype system. We offered the participants the option to either plug in their keyboard or use the keyboard of our prototype. All but one participants opted to use our keyboard, as it felt more convenient for them not to unplug their keyboard, or because they were using laptops and docking stations.

We asked the participants to type the same, randomly-selected paragraph of the Bacon Ipsum text comprising 71 characters. Figure 3 depicts the distribution of the distance between each of the 71 recorded events (i.e., the keypresses interarrival time) produced by each participant in box-and-whisker diagrams. The diagrams indicate that each participant exhibited different typing patterns. Almost all diagrams contain a large number of outliers towards larger numbers. Delays of one or even three seconds between keypresses are noticed. The overall median value was 0.20 seconds and the average time reached 0.30 seconds. Despite the per-participant differences, there was *not a single case* where our research prototype detected erroneously a RES, i.e., once more, there were *zero false positives*.

Our typing dynamics analysis results are in alignment with published literature [15]. Hence and for the sake of research efficiency and economy, we opted not to expand the study to more participants.

The second part of the evaluation confirmed that typing dynamics are quite stable among users, despite some unavoidable differentiations. More importantly, even in short texts, the human typing dynamics are clearly above the detection threshold we have defined and clearly distinguishable from that of the Rubber

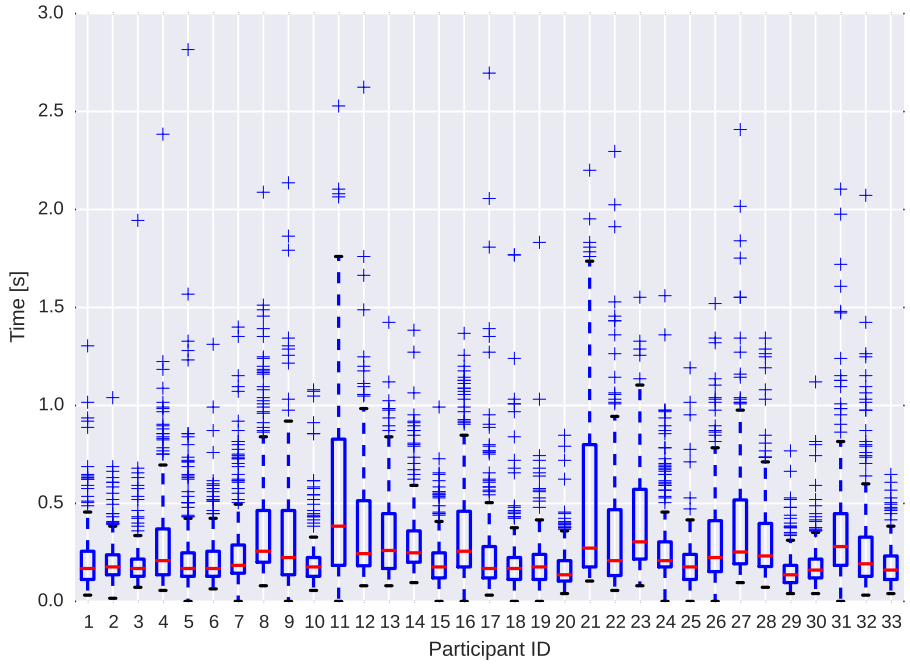


Fig. 3: Temporal variations in the typing dynamics experiment.

Ducky and BadUSB attacks (cf. Figures 3 and 2). Thus, such information can serve as a heuristic for detecting and defending against the attacks.

5 Conclusions and future work

BadUSB-like attacks are a realistic threat. Yet, system-level defenses cannot be realized in the form of malware analysis tools for USB firmware. USB device whitelisting can offer some protection in specific usage scenarios, but its usability is hindered when scaling in enterprise-level networks. Proposed defenses in the literature mandate the user involvement in the trust decisions. This is a suboptimal, error-prone design choice.

In this paper, we studied the *temporal* characteristics of BadUSB-like attacks. We proposed USBlock to block malicious USB packet traffic. Our proposal is extensible and can integrate additional features for coping with future, advanced attacks. Residing on the side of the operating system, USBlock has an advantage over a malicious USB firmware payload executing on a peripheral device and interacting with the main system.

We implemented a proof-of-concept defense module for the Linux kernel. We evaluated its stability under different usage patterns for three months and studied the user temporal variations and typing dynamics in a small-scale study.

The collected evidence suggest that our implementation caused no issues while human typing behavior was clearly distinguishable from that of the existing known attacks.

Our findings indicate that it is feasible to realize advanced defense mechanisms for BadUSB-like attacks by integrating system-level temporal characteristics and *without* involving the user in the trust decisions.

Further validation of our findings by the network and system security community can provide stronger confidence for the applicability of our approach. We envision that such information as well as information regarding the spread of USB devices across an enterprise network can be used as additional feature in order to enrich the malware detection process and enhance our arsenal in fighting cybercrime.

Acknowledgements

This research was supported by the Austrian Research Promotion Agency (FFG) through the BRIDGE 1 grant P846070 (SpeedFor) and the COMET K1-Centres programme line (SBA2). S. Neuner was also supported by the Austrian Marshall Plan Foundation through a Marshall Plan Scholarship. We thank Prof. E. Kirda and W. Robertson for their valuable support during the early stages of this research as well as the participants in the typing experiments.

References

1. Anderson, B., Anderson, B.: Seven deadliest USB attacks. Syngress (2010)
2. Angel, S., Wahby, R.S., Howald, M., Leners, J.B., Spilo, M., Sun, Z., Blumberg, A.J., Walfish, M.: Defending against malicious peripherals with Cinch. In: USENIX Security Symposium (2016)
3. Dingledine, R., Mathewson, N.: Anonymity loves company: Usability and the network effect. In: Workshop on the Economics of Information Security (WEIS) (2006)
4. Fidas, C., Voyiatzis, A., Avouris, N.: When security meets usability: A user-centric approach on a crossroads priority problem. In: 14th Panhellenic Conference on Informatics (PCI 2010) (2010), tripoli, Greece, September 10–12, 2010.
5. Griscioli, F., Pizzonia, M., Sacchetti, M.: USBCheckIn: Preventing BadUSB attacks by forcing human-device interaction. In: Privacy, Security and Trust (PST), 2016 14th Annual Conference on. pp. 493–496. IEEE (2016)
6. Guri, M., Kachlon, A., Hasson, O., Kedma, G., Mirsky, Y., Elovici, Y.: GSMem: Data exfiltration from air-gapped computers over GSM frequencies. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 849–864 (2015)
7. Han, S., Shin, W., Kang, J., Park, J.H., Kim, H., Park, E., Ryou, J.C.: IRON-HID: Create your own bad USB (white paper). HITBSecConf 2016 - Amsterdam. The 7th Annual HITB Security Conference in the Netherlands (2016)
8. Johnson, P., Bratus, S., Smith, S.: Protecting against malicious bits on the wire: Automatically generating a USB protocol parser for a production kernel. In: Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017. pp. 528–541. ACM (2017)

9. Kang, M.: USBWall: A Novel Security Mechanism to Protect Against Maliciously Reprogrammed USB Devices. Master's thesis, University of Kansas (2015)
10. Langner, R.: Stuxnet: Dissecting a cyberwarfare weapon. Security & Privacy, IEEE **9**(3), 49–51 (2011)
11. Loe, E.L., Hsiao, H.C., Kim, T.H.J., Lee, S.C., Cheng, S.M.: Sandusb: An installation-free sandbox for usb peripherals. In: Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on. pp. 621–626. IEEE (2016)
12. Neugschwandtner, M., Beitler, A., Kurmus, A.: A Transparent Defense Against USB Eavesdropping Attacks. In: Proceedings of the 9th European Workshop on System Security. pp. 6:1–6:6. EuroSec '16, ACM (2016)
13. Nissim, N., Yahalom, R., Elovici, Y.: USB-based attacks. Computers & Security **70**(Supplement C), 675–688 (2017)
14. Pham, D.V., Syed, A., Halgamuge, M.N.: Universal serial bus based software attacks and protection solutions. Digital Investigation **7**(3), 172–184 (2011)
15. Teh, P.S., Teoh, A.B.J., Yue, S.: A survey of keystroke dynamics biometrics. The Scientific World Journal **2013** (2013)
16. Tian, D.J., Bates, A., Butler, K.: Defending Against Malicious USB Firmware with GoodUSB. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 261–270. ACM (2015)
17. Tian, D.J., Bates, A., Butler, K.R., Rangaswami, R.: ProvUSB: Block-level provenance-based data protection for USB storage devices. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). pp. 242–253. ACM, New York, NY, USA (2016)
18. Tian, D.J., Scaife, N., Bates, A., Butler, K., Traynor, P.: Making USB great again with USBFILTER. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 415–430. USENIX Association, Austin, TX (2016)
19. Tischer, M., Durumeric, Z., Foster, S., Duan, S., Mori, A., Bursztein, E., Bailey, M.: Users Really Do Plug in USB Drives They Find. In: Security and Privacy (SP), 2016 IEEE Symposium on. IEEE (2016)
20. Umphress, D., Williams, G.: Identity verification through keyboard characteristics. International journal of man-machine studies **23**(3), 263–273 (1985)
21. Wang, Z., Stavrou, A.: Exploiting smart-phone USB connectivity for fun and profit. In: Proceedings of the 26th Annual Computer Security Applications Conference. pp. 357–366. ACM (2010)
22. Yang, B., Feng, D., Qin, Y., Zhang, Y., Wang, W.: TMSUI: A trust management scheme of USB storage devices for industrial control systems. In: Information and Communications Security: 17th International Conference, ICICS 2015, Beijing, China, December 9-11, 2015, Revised Selected Papers, pp. 152–168. Springer International Publishing, Cham (2016)
23. Zaitcev, P.: The usbmon: USB monitoring framework. In: Linux Symposium. p. 291 (2005)