USBCheckIn: Preventing BadUSB Attacks by Forcing Human-Device Interaction

Federico Griscioli*, Maurizio Pizzonia* and Marco Sacchetti*
*Roma Tre University, Department of Engineering
Via della Vasca Navale 79, 00146 Rome, Italy
{griscioli,pizzonia}@dia.uniroma3.it m.sacchetti@quipo.it

***Abstract*—The BadUSB attack leverages the modification of firmware of USB devices in order to mimic the behaviour of a keyboard or a mouse and send malicious commands to the host. This is a new and dreadful threat for any organization. Current countermeasures either require special USB devices or ask the user to decide if the device can be used.**

We propose a new approach that, before allowing the device to be used, forces the user to interact with it physically, to ensure that a *real* human-interface device is attached. Our implementation is hardware-based and, hence, can be used with any host, comprising embedded devices, and also during boot, i.e., before any operating system is running. Our approach does not require any special feature from USB devices.

I. Introduction

Traditionally, USB security mostly deals with thumb drives and their role as infection vector or as vehicle for confidential information leakage. Recently, the presentation of the BadUSB [1] class of attacks disclosed new threats that involve USB. These attacks are based on a modification of the device firmware, that forces the infected device, typically a thumb drive, to behave as a different kind of device, for example as a keyboard. A malicious "virtual" keyboard can inject commands that end up in a malware infecting the host. These malicious commands could download a malware from the Internet, but can also create it on-the-fly, for example, by "typing" the content of a malicious script and running it. In this context, regular antiviruses are largely ineffective, since the malicious USB device is exploiting basic capabilities (e.g., typing operating system commands) that the user is normally allowed to use.

The primary countermeasure proposed against BadUSB was to *protect* USB devices by a firmware authentication feature that limits the firmwares that can be uploaded into a device to those signed by the device vendor (see for example [2]). However, this approach protects devices but not hosts, which are secured only if they are forcibly limited to interact with just protected devices. This strongly limits the usability of USB devices and it is insecure, if the limitation is delegated to error-prone user behaviour.

GoodUSB [3] is a software solution that aims at protecting the host against BadUSB attacks. When a new USB device is attached, a message is shown to the user, which must declare his/her expectation about the functionalities of the device.

In this paper we present USBCheckIn, an hardware solution that is able to protect any kind of USB host against attacks from devices that claim to be *human interface devices* but are not. The basic idea is that the authenticity of a real human interface device can be easily checked by asking the user to use it. To *authorize* a human intreface device to connect to the host, USBCheckIn instructs the user to perform certain actions on the human interface device by showing messages on its own display. This makes it completely independent from host capabilities and, hence, compatible with any kind of host. While devising USBCheckIn, we focused on verification of keyboards and mice, and we designed the human-machine interaction for high usability (e.g., just 3 gestures to verify a mouse), while preserving security and keeping the probability of a successful brute-force attack negligible.

USBCheckIn has several advantages with respect to other state-of-the-art proposals. (1) It requires a human to provide a proof that the device is, indeed, a human interface device. This ensures that even naive or malicious users cannot behave in a way that the attack is successful. (2) It is compatible with any kind of USB hosts, USB devices, operating systems, BIOS, and boot loaders. (3) It does not rely on signatures, heuristics, or parameters that are provided by the device.

Our contributions are the following: (1) we describe the architecture of USBCheckIn (Section III), (2) we show the interaction between USBCheckIn, the user, and the device that leads USBCheckIn to determine if the device is indeed a real human interface device (Section IV), (3) we provide a security analysis of our approach (Section V). In Section VI, conclusions are drawn.

A *companion video*, showing a prototypical realization of USBCheckIn, can be downloaded from the Internet [4].

II. Background

In this section we briefly recall some of the basic concepts about the USB protocol.

An actor in the USB communication can have one of two roles: *device* or *host*. When a device is plugged into the USB port of an host system, the host performs an *enumeration*. During this stage, the host discovers the *class* of the device and its *type*. This paper mainly deals with devices belonging to the *Human Interface Device* (HID) class as defined by the USB standard, in particular, of *type* mouse and keyboard[1]. Other device classes refer, for example, to mass storage, printing, and audio/video streaming. HIDs are used to submit human input to the system and hence are fundamental in the BadUSB attack. In this paper, we focus on the authorization of mice and keyboards while inhibiting the use of other HIDs, like joysticks. After enumeration has been performed by the host, normally the device can be used until it is disconnected. The device can simulate a physical disconnection from the host by sending a *detach* signal. This allows the device to trigger a new enumeration and present a class and a type different from the previous enumeration. Indeed, a physical USB device can be composite and can show to the host several *interfaces*, each of them with a distinct $\langle class, type \rangle$ pair. For the purpose of this paper, they can be considered distinct devices, since, the communication channels from the host toward each interface of the device are completely independent, hence, any filtering can be selectively performed. A device never speaks autonomously (beside the detach signal case). The protocol states that the host periodically polls all the devices (interfaces), which must start replying within a very short time, compared to frame length. Timing is so strict that a malicious reply from a non-polled device gets mixed with the genuine answer and discarded. The host does not poll the next device until either receive a reply or a timeout expires. Reply packets have no source field, but since they can only follow a poll from the host, their source is easily deduced.

III. Architecture

In this section we present the architecture of USBCheckIn, as shown in Figure 1. USBCheckIn is equipped with two USB ports: one *upstream port* and one *downstream port* (*up/down ports*). The up port of USBCheckIn is connected to one of the USB ports of the host system we intend to protect. For full protection, we assume that all other USB ports of the host are not used (e.g., they might be physically disabled) and all devices the user intends to attach to the host are actually attached, possibly through a hub, to a down port. For the sake of simplicity, we present the architecture of USBCheckIn with only one down port. The handling of additional down ports is a straightforward extension of the approach proposed in this paper. Near the up port, there is an orange LED that indicates that the host is powering USBCheckIn and it is correctly working (e.g., it is not suspended). Near the down port, there is a status red/green LED. The LED blinks green when one of the attached devices, possibly through a hub, is undergoing the authorization procedure, turns fixed green when all devices are successfully authorized and can interact with the host, and turns blinking red if the authorization procedure of one of the attached devices failed too many times and the device must be disconnected. Messages required for user interaction are

[1]Actually, with the word "type" we refer to the value of the bInterfaceProtocol field in the USB standard.

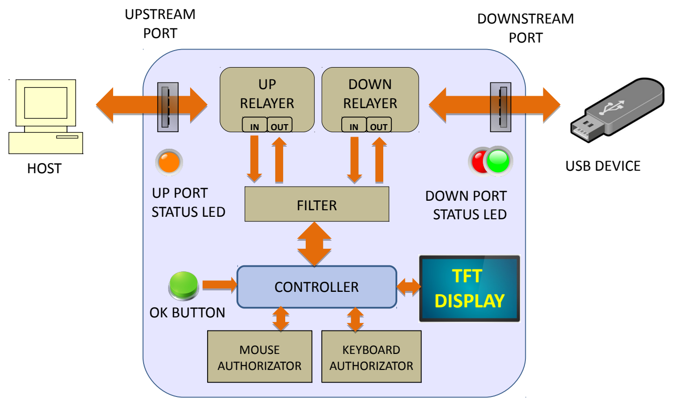

Fig. 1. The architecture of USBCheckIn.

presented to the user on a 2.1" TFT display which is embedded in USBCheckIn. An "ok" button is also present, since a direct interaction is needed for certain special cases (see Section IV).

USBCheckIn is based on the Beaglebone Black board [5] on which a Linux kernel is running. In USBCheckIn, a software, running in user space, intercepts, inspects, redirects, and injects USB traffic between up and down ports, depending on the state of the authorization procedure, as described in Section IV. The USBCheckIn software is made up of several components.

Relayers. On one side, up/down relayers are responsible of sending and receiving packets to and from the USB ports. On the other side, packets are forwarded to and received from the filter.

Filter. The filter is in charge of performing standard USB initialization steps (like enumeration, see Section II) and passes USB packets to and from the relayers and the controller depending on the authorization status of the device as explained in Section IV.

Controller. The controller orchestrates the filter, the display, the status leds and the authorizator to realize the authorization procedure and human-machine interaction described in Section IV.

Authorizators. Authorizators manage the authorization process of HIDs connected a down port, by generating challenge codes, keeping track of the number of attempts, and checking the correctness of the submitted code (see Section IV). Each authorizator corresponds to a specific type of HID. The correct authorizator is chosen by the controller according to the type declared by the device.

The software running on USBCheckIn is a customized version of USBProxy [6]. The filter and the authorizators are USBProxy plugins. The communication between the device connected to a down port and the down relayer is realized by means of libUSB [7], a library that supports the interaction with generic USB devices, while the communication between the host and the up relayer is realized by means of gadgetFS, a linux kernel module that allows the Beaglebone to act as a client towards the host.

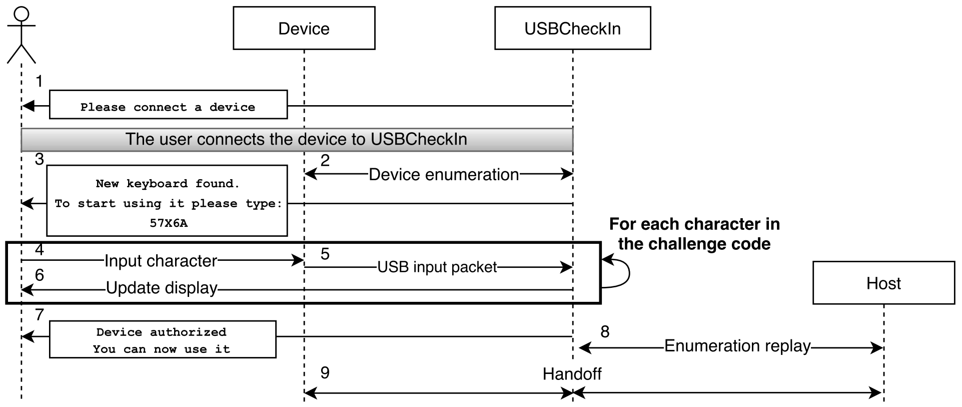

Fig. 2. Interactions and messages among user, device, USBCheckIn, and host. The diagram shows messages for the keyboard authorization procedure.

IV. Interactions

In this section, we describe the interactions between four actors: a human, USBCheckIn, the USB device just plugged, and the host. In the absence of USBCheckIn, when a device is directly attached to a host, the host enumerates the device (see Section II). When USBCheckIn is attached to the host, it prompts the user to connect a device by showing a proper message on the display (see Figure 2, step 1, and the companion video [4]). When the user attaches the device to USBCheckIn, the latter performs the enumeration (step 2). In this phase, USBCheckIn recognizes the capabilities of the device and records all provided data.

If the device is an HID, USBCheckIn starts the authorization process asking to the user to input, **by means of the HID itself**, a randomly-generated challenge code (Step 3).

In case of a keyboard the challenge code is a 5-characters alphanumeric string, which is communicated to the user by showing on the display a message like the following

```
New keyboard found
To start using it please type:
57X6A
```

The user types/clicks the *elements* of the challenge code (Step 4). For each of them, the element (for a keyboard is a key code, for a mouse is a click event) is sent by the device to USBCheckIn according to HID normal behaviour and the regular USB protocol (Step 5). The authorizator checks that each received element matches with the corresponding element of the challenge code. If it matches, USBCheckIn provides a visual feedback to the user (Step 6). For example, for a keyboard, by coloring green the corresponding element on the display. Then, USBCheckIn loops expecting further elements and checking them until the challenge code is finished.

If the user correctly inputs all challenge elements, USBCheckIn notifies her/him that the new device has been authorized and that it can be used (step 7). At the same time, USBCheckIn impersonates the device toward the host by performing a "replay" of the enumeration that was previously recorded (step 8). Then USBCheckIn performs an *handoff* by configuring the filter to short-circuit the up relayer and the down relayer (step 9). USBCheckIn keeps monitoring the exchanged packets between the host and the device in order to recognize hardware or logic *detach/re-attach* operations and then trigger a new authorization procedure, when needed.

If any of the received elements does not match the corresponding element of the challenge code, the authorization is started over again showing a message that, for a keyboard, are like the following.

```
Wrong code - try again.
To start using the keyboard pls type:
7E5N3
```

The authorization process can be tried (steps 3-6) for a maximum of three times. After three wrong attempts, USBCheckIn definitely blocks the device, ignoring any more input from it, and shows a message on the display, as follows:

```
    *** Authorization failed ***
 Device claims to be a [device type].
 Is it true? Is the device malicious?
To check it again, press the ok button.
```

In this message, "[device type]" can be "keyboard" or "mouse". To try again, the user has to push the "ok" button on the USBCheckIn hardware, then the authorization procedure starts over from step 2. This physical interaction makes a completely automatic brute-force attack impossible. This message is displayed also when a malicious device tries a guess attack.

If the device is a mouse, USBCheckIn asks the user to move a pointer on the display to draw a line between two randomly-placed targets, for 3 times. The procedure was selected to obtain a good compromise between security and usability. In this case, each element of the challenge is a pair of points of the display, and the input of the user matches the element of the challenge if and only if the distance between the points of the challenge and the clicked ones is below a certain *radius*. The radius is chosen so that 24 non-overlapping targets can fit in the display (see Figure 3). Instruction messages for the user are adaptively placed on the display on an unused zone to allow higher freedom in target positioning. See section V for a security analysis. If the class of the device is not HID,

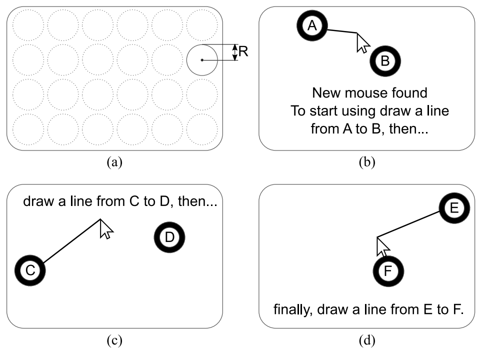
R
A
B
New mouse found
To start using draw a line
from A to B, then...
draw a line from C to D, then...
C
D
E
F
finally, draw a line from E to F.
(a)
(b)
(c)
(d)

Fig. 3. The mouse authorization method. (a) Allowed target positions and radius R. (b,c,d) The three steps of the authorization: in (a) just two rows are available for second target positioning, in (b) and (c) three rows are available.

the display shows to the user the capabilities declared by the device with a message like the following:

```
The connected device claims to be:
   a webcam
To authorize it, hold down OK for 2 secs.
```

Once the message is displayed, the user has to authorize the device explicitly, by means of long-pressing the button located on USBCheckIn. If the user allows the device, USBCheckIn replays the enumeration toward the host and the filter is programmed to short-circuit the relayers for that USB traffic. After that, the device can directly communicate with the host. To deny the device, the user just disconnects it.

V. Security analysis

In this section, we discuss the effectiveness of USBCheckIn in protecting any host from BadUSB attacks that mimic HID behaviour, for example performing keystroke injection.

Our approach is not vulnerable to human mistakes. In fact, we do not ask to the user to take any decision, like accepting device features or choosing in a list of allowed device classes (but for the case of non-HID devices, which are not the primary target of this work). Actually, not even a malicious user can force USBCheckIn to authorize a device that claims to be an HID but offers no means to the user to provide the requested challenge code.

Our approach is extremely well protected against guessing/brute-force attacks. For keyboard authorization, the challenge code consists of 5 characters each ranging within 26 letters plus 10 digits. The probability for a malicious device to correctly guess a random challenge in three attempts is 3 over 36^5 (i.e., 1 over about 20 millions). For mouse authorization, each challenge code consists of 3 pairs of points. The first point of each pair ranges within 24 possible positions. The second point ranges within the unused positions that remain after the text message is placed: 11 for the first elements (Figure 3b) and 17 for the second and third element (Figures 3c and 3d). The probability of a successful attack in 3 attempts is 3 over $24^3 \cdot 11 \cdot 17^2$ (i.e., 1 over about 14 millions).

It is worth noting that a malicious device has no clue about the success or failure of each attempt and after three failed attempts the human intervention is required to gain more attempts. Non-HID devices cannot maliciously mimic HID behaviour: once a device is authorized as non-HID, any attempt to mimic HID behaviour requires a re-enumeration that can be triggered by a detach signal, which in turn triggers a new authorization process by USBCheckIn.

The above analysis assumes that USBCheckIn has no security bugs and it is not compromised. While this is a demanding assumption, honoring it for a dedicated hardware is surely easier than honoring it for a software which runs on the host. In fact, the security of USBCheckIn is based on the security of a reasonably small and stable amount of software (the software we developed, USBProxy, libUSB, and the kernel), while for a host-based solution all software running on the machine should be trusted unless proper mandatory access control configurations are in place, which may not be feasible in many environments. Also, certification procedures are much easier for an isolated system.

Finally, we note that our approach has an obvious limit: it cannot prevent a *malicious HID* to actually allow the user to enter the challenge code. However, other proposals, like for example [3], have similar limitations and, to our knowledge, this is still an open problem.

VI. Conclusions

We presented USBCheckIn, a hardware that realizes a new approach to protect hosts from BadUSB attacks in which generic USB devices maliciously mimic the behavior of HID devices. Our proposal does not relies on decisions of the users and it is compatible with any host or device. Our first experiments with our prototypical implementation show that the presence of USBCheckIn has no impact on HID devices responsiveness (e.g., mouse polling frequency is the same) and CPU consumption is unnoticeable. Future plans encompass performing deep performance tests and a formal user study for assessing the usability in real environments.

VII. Acknowledgements

The authors are grateful to Leonardo Silvi for his work on the BadUSB attack. This work has been partially supported by the European Union Seventh Framework Programme under grant agreement no. 607093 “Preemptive” and by MIUR Project AMANDA under PRIN 2012C4E3KT.

References

[1] K. N. . J. Lell, “BadUSB - On Accessories that Turn Evil,” https://www.blackhat.com/us-14/briefings.html#Nohl, Blackhat USA, [Online; accessed 27-July-2016].

[2] IRONKEY^{TM}, “Secure USB Devices: Protect Against BadUSB Malware,” http://www.ironkey.com/en-US/solutions/protect-against-badusb.html, [Online; accessed 27-July-2016].

[3] D. J. Tian, A. Bates, and K. Butler, “Defending against malicious usb firmware with goodusb,” in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 261–270.

[4] “**USBCheckIn companion website and video**,” https://bitbucket.org/sec3/usbcheckin.

[5] beagleboard.org, “BeagleBone Black,” https://beagleboard.org/black, [Online; accessed 27-July-2016].

[6] D. Spill, “USBProxy,” https://github.com/dominicgs/USBProxy, [Online; accessed 27-July-2016].

[7] “libusb: A cross-platform user library to access USB devices,” http://libusb.info/, [Online; accessed 28-July-2016].