Using Context and Provenance to defend against USB-borne attacks

Tobias Mueller
mueller@informatik.uni-hamburg.de
University of Hamburg, Germany

Ephraim Zimmer
ezimmer@informatik.uni-hamburg.de
University of Hamburg, Germany

Ludovico de Nittis
denittis@gnome.org
GNOME

ABSTRACT

Today's readily available security measures to defend one's computers against malicious USB devices either show popups that require the user to allow each interaction, or they use identity-based peripheral devices attachment rules to allow or deny interaction with the new USB device, which again involves the user. In this paper, we propose a novel strategy for defending against USB attacks with the main goal of not involving the user.

For making the security relevant decision, we take both context of the user's session and provenance of the security relevant event into account. That is, we assume that the user cannot plug a device into their machine when they are not present, e.g. when they have left their computer. We infer that the state of the lock screen relates to the presence of the user and do not allow new USB devices when the screen is locked. Further, we deflect traditional BadUSB attacks by taking the provenance of dangerous keystrokes into account when making an automated security decision. We extend the same idea to other security relevant contexts, such as network re-configuration.

To substantiate our claims, we identify two classes of USB-borne attacks: driver exploitation and user emulation. While the first exploits could and can be prevented with secure coding and runtime mitigations, the second does not circumvent bugs in code but rather masquerades a device as another. We also investigate real-world usage of USB and present data which shows that we can expect users to have a single keyboard. Consequently, we increase protection against said masquerading attacks by filtering keys deemed dangerous or preventing security relevant actions if the keystroke originated from a newly attached USB device. We present an implementation of our filter for both GNU/Linux and Microsoft Windows.

CCS CONCEPTS

• **Security and privacy → Usability in security and privacy**; *Operating systems security*; Hardware attacks and countermeasures; • **Human-centered computing** → *Empirical studies in HCI*.

KEYWORDS

USB attacks, usability, usable security

ACM Reference Format:
Tobias Mueller, Ephraim Zimmer, and Ludovico de Nittis. 2019. Using Context and Provenance to defend against USB-borne attacks. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES 2019) (ARES '19), August 26–29, 2019, Canterbury, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3339252.3339268

1 INTRODUCTION

The Universal Serial Bus (USB) is a very popular standard for interfacing with external devices. It is in widespread use and thus exposes a compelling attack surface [17, 23, 26, 27, 36]. In fact, famous attacks such as "Stuxnet" were delivered via USB [13]. Without a fundamental change in the concept and infrastructure of USB, defences against USB-borne attacks can arguably not reach full coverage [36]. The reasons are manifold. Research by Nohl and Lell [27] showed, for example, that USB device controller is not authenticated by an operating system, but only identified with information that is provided by the device controller itself, which means a device can identify itself as several different device types and even change the device type while staying plugged in. So a harmless looking but specially prepared USB mass storage device can re-identify itself as a new keyboard to the operating system at any time and issue key combinations and commands, which could infect the machine where the device is plugged in. Additionally, the device controller of a benign USB device is generally not protected against reprogramming, which means an infected machine can easily reprogram any USB device that is connected, even without leaving any sign to a user. These two attack scenarios alone already establish a devastating infection and spreading strategy of any kind of malware and are inherent to the USB infrastructure and concept. Although various defences have been proposed in the past [2, 12, 14, 16, 33, 34] in order to mitigate these attack scenarios, none of them have reached widespread acceptance, nor have been generally available as a built-in mechanism in any operating system of the user.

This paper describes the design and Open Source implementation of a baseline mitigation technique against attacks coming from malicious USB devices, which follows *Security-by-Design* patterns [11] and thereby minimises the need of user interaction as much as possible while making security and trust relevant decisions regarding USB devices. This approach results in a more usable defence mechanism and can directly be integrated into the USB stack of all operating systems. By tightly integrating our protection mechanism, stronger protection capabilities and a better coverage can be achieved compared to existing tools, as no additional protection mechanism has to be installed and configured by the user and the integrated protection mechanism is active and running as early as possible during the system startup.

In Sect. 2 we provide background knowledge about USB. Sect. 3 provides a classification of USB-borne attacks in order to derive basic attack principles and structural problems and misconstructions of USB. This includes deduction of typical real-life usage of USB devices in order to prevent or at least exacerbate atypical USB device behaviour. In Sect. 4 and Sect. 5 follows our main contribution, the design and open source implementation of an non-intrusive baseline defence mechanism. Thereby we structurally mitigate unsophisticated USB-borne attacks that aim for defence-less systems. In Sect. 6 we give an overview of other approaches on USB protection mechanisms. This paper concludes with a discussion in Sect. 7.

2 BACKGROUND

This section describes the main properties of USB as well as the preliminaries of our defence mechanism.

2.1 Universal Serial Bus

USB was designed to standardise the connection of peripherals like keyboards, printers, disk drives etc. to personal computers, allowing them to communicate as well as supplying electric power [3, 10]. A computer's USB port can be extended with a "hub", which is a special USB device and can accept multiple other USB devices. USB allows devices to be composed of several *functions* which can be thought of as services provided to the user, e.g. mass storage or keyboard. A USB device can be compound or composite in order to implement multiple functions. This allows users to use only one physical device, but get multiple functionalities. An example is a cordless input arrangement consisting of a keyboard and a mouse with only one physical plug to be inserted into the computer. A compound device contains an internal hub to which other internal devices are then connected, each receiving its own address, which is used for the subsequent communication over the bus. A composite device has only one address on the bus but multiple interfaces, one for each provided function. Either way, when a USB device is attached to a host, the operating system enumerates the new device. That process causes the device to send either one or a set of interface *descriptors* which represent the protocol of the messages the device expects and what functions it provides. The operating system uses this information to load the appropriate driver for each interface, sets its configuration accordingly and then the USB device can start its normal operation.

A recent analysis of all USB specifications including USB 3.x shows, that security has not been taken into consideration [36]. In fact, the USB consortium considered ensuring the security of a USB device a problem of the user [39]. This has changed with the relatively recent specification of USB Type-C Authentication [38] which uses a Certificate Authority Model comparable to the one used for TLS in order to authenticate certain certified USB devices. This specification has neither reached widespread adaption nor does it effectively protect against USB-borne attacks, as it does not include the firmware during the verification process [36].

2.2 USB as Attack Vector

USB's versatility leads to a worrying problem: because different device classes use the same connector type, one device can masquerade as benign device to the user but actually identify as another device to the operating system. This identification to the operating system can even be changed at any time while the USB device is plugged into a host. USB allows a de-registration and re-registration without the need to physically unplug the device. The device then gains different or additional capabilities, which the user did not expect. This has been demonstrated multiple times in the past, e.g. through the arguably most prominent attack coined "BadUSB" [27].

In order to turn one device type into another, a USB controller chip can be reprogrammed. A lot of controller chips, like the ones used for mass storage devices (or "thumb drives"), have no protection from such reprogramming [16, 27].

With such a reprogrammed device, it is for example possible to:

- Transform a USB stick into a computer keyboard and issue commands on behalf of the logged-in user [27].
- Transform a USB stick into a keylogger and extract passwords typed in by the user [24, 32].
- Transform a USB stick into a network card and change the DNS setting to redirect traffic [18].

As USB can be used to attach pretty much any device to the computer, it can be used to interact with many of the installed device drivers. Those drivers are often of questionable quality, which increases the risk of successful attacks [8, 9, 15]. At the time of writing more than 250 CVEs are assigned for issues revolving around USB.[1]

Academia has investigated the phenomenon of USB-borne attacks. Nissim et al. identified 29 types of USB-based attacks [26]. Ranging from the oldest and most famous Rubber Ducky[2], that emulates a keyboard and injects a pre-loaded keystroke sequence, to the more recent and sophisticated like the TURNIPSCHOOL[3], that provides short range RF communication capability to software running on the host computer.

2.3 USB as Trust Anchor

The typical usage of USB devices breaks down what usually is considered as trust barriers from a security point of view. The host machine of a user enjoys special protection by all kinds of different security mechanisms and users are trained to pay special attention to the security of their machines. These efforts are necessary, because the host of a user constitutes a trust anchor. Once the own machine is infected, most if not all security mechanisms can be subverted. So several barriers are build to protect this trust anchor. USB however is not considered a potential security threat, as research by Tischer et al. [37] has shown. They dropped nearly 300 infected USB sticks outside of a parking lot and monitored, how many USB sticks have been picked up and connected to a computer by unwitting people. The authors estimated a success rate of 45-98% with the first connected drive that called home in less than six minutes.

[1]cf. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=usb
[2]https://github.com/hak5darren/USB-Rubber-Ducky
[3]http://www.nsaplayset.org/turnipschool

Consequently, it can be concluded, that the USB ecosystem is considered part of that trust anchor too, even though a USB device might potentially be under control of an attacker or an USB device owned by a user is plugged into a machine, that might potentially be under control of an attacker.

2.4 Security-by-Design

Research has shown that users tend to cognitively dismiss pop-ups [6]. The authors further suggest that "pop-ups are similar to annoying or rude colleagues who interrupt one's current task and insist on interaction" [6] and that a security solution should try to "not interrupt the user while the user is engaged in an ongoing task" [6]. This is in line with a study showing that users go to great length, even putting their computer at risk, in order to perform a task [30]. The reasons, which prevent users from understanding security warnings are manifold [41], among which the lack of integration with the existing metaphors of the desktop GUI has been identified as a problem [31]. Likewise, users and their understanding of security relevant dialogues was investigated with the insight that interrupting a user while performing a task is detrimental to getting an informed decision, also, because users either just uninstall or disable the security software or get habituated to the prompts [29]. Security software needs a user centred design as well as knowledge about how users will actually use the designed solution [1]. Successful security software should "design security into all of an application's layers (in particular, its upper layers)" in order for security to "become implicit and hence much more user-friendly" [7]. It is recommended to "seek ways to extract and use as much accurate information as possible from a user's normal interactions with the interface" in order to align security and usability [40]. We work to that end by inferring whether the user is present and infer whether the user's mental model includes being able to attach a new USB. This approach is supported by the insight that users rarely interact with a computer system to perform a security task [5]. Rather, they have a primary goal such as socialising with their friends, writing a document, or sending files. As such, any interaction the user has to perform to fulfil a secondary task, such as authorising a new USB device, distracts from the primary task and is recognised as annoyance [6].

We note that is has been argued that "taking the end user out of the [security] loop does not solve the problem" [5]. However, we do not aim for maximising protection, but for maximising protection for most users in most circumstances. To that end, we acknowledge that our approach does not protect against a sophisticated and targeted attack.

3 ATTACKING MACHINES VIA USB

In this section we first explain the underlying threat model, which will be considered throughout the rest of the paper. Subsequently, we present our findings regarding the factors that enable USB-borne attacks.

3.1 Threat Model

For this work we assume that the host operating system can be exposed to newly attached USB devices at any time. Note that this carefully excludes the phase in which the operating system is not yet fully booted, i.e. the BIOS or early boot phase. We further assume that the host operating system is not actively malicious, e.g. that the drivers for USB devices were not written for the purpose of allowing exploitation from a devices attached via USB. Rather, we assume that exploitability of drivers stems from mistakes during implementation. USB devices, on the other hand, can behave in any way on the protocol layer which defines how a host and a device communicate. However, we exclude misbehaviour on the lower two layers of the USB protocol, which are the physical layer responsible for the electrical specifications and the link layer responsible for, e.g. addressing [3, 10]. Hence, we exclude attacks working on the electrical level, e.g. working with too high voltages[4].

Our model allows for an attacker to modify the functionality as well as the appearance of a USB device and physically plug it into a victim's machine by themself or by tricking the victim to physically connect the malicious USB device to their machine.

3.2 Classification of USB based Attacks

The phenomenon of USB-borne attacks has been investigated in the past [26, 36]. For our purposes and in accordance with our threat model, we classify attacks via the USB as either exploiting a vulnerable driver or simulating user behaviour.

Our classification is deliberately simple to allow the deduction of basic principles, which need to be deployed in order to mitigate USB based attacks.

3.2.1 Vulnerable Driver. This class of attacks is made possible by insecure coding and can have devastating results if the compromised driver lives in kernel-space. Although USB drivers can be written in user-space, the majority of drivers, especially generic ones, live in kernel-space. A malicious device can thus avail of the whole operating system once it has exploited a driver [23].

We note that this class can be eliminated by isolating drivers, e.g [8, 9, 15], but popular operating systems such as Windows or GNU/Linux do not follow a micro-kernel design, which does not make isolation of drivers impossible, but harder to deploy.

3.2.2 Simulating User Behaviour. This class of attack involves a device with different capabilities than the user expects. Instead of exploiting vulnerable code, the device exploits user expectation. For example, the BadUSB attack relies on the user expecting a mass storage device but the device is acting as a keyboard. This attack has come in various flavours, including devices which act as a network card and then wiretap all connections of the machine; even when the screen is locked. Note that no vulnerable code of the operating system is exploited as the attack works on a higher level. Securing single layers rather than cross-layer has been identified as limiting the usefulness of protection mechanisms [36].

3.3 Structural Problems

Several areas can be identified, which enable a malicious USB device to become a problem.

3.3.1 USB is Always On. The USB interface of a computer is constantly enabled and waiting for a device connection. This means,

[4]e.g. USBKill: https://usbkill.com/

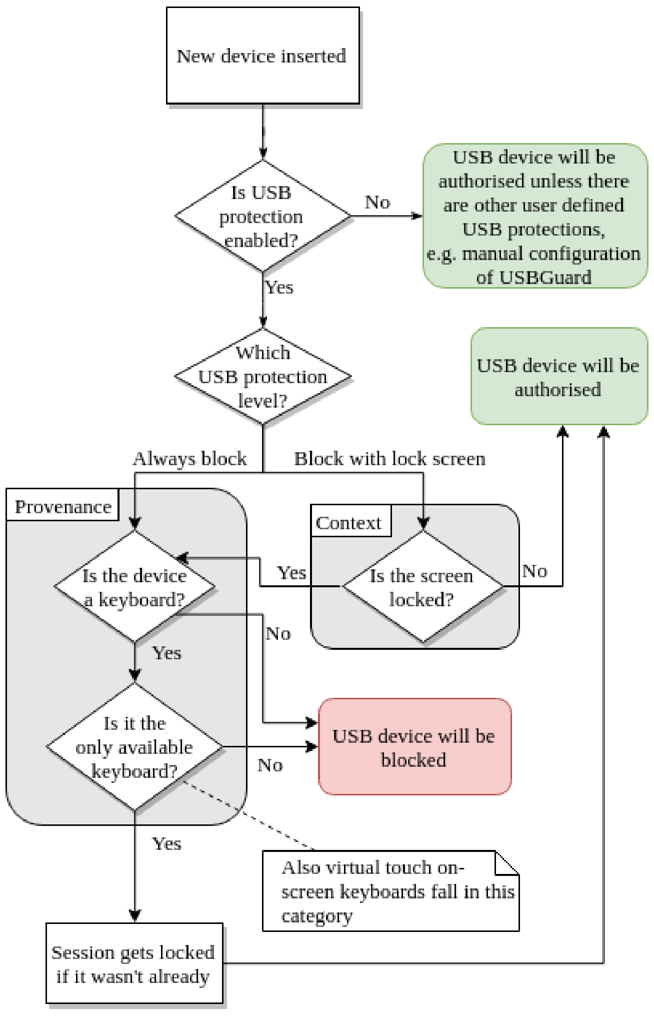

New device inserted
Is USB protection enabled?
No
USB device will be authorised unless there are other user defined USB protections, e.g. manual configuration of USBGuard
Yes
Which USB protection level?
USB device will be authorised
Always block
Block with lock screen
Provenance
Context
Is the device a keyboard?
Yes
Is the screen locked?
No
No
Yes
Is it the only available keyboard?
No
USB device will be blocked
Yes
Also virtual touch on-screen keyboards fall in this category
Session gets locked if it wasn't already

Figure 1: Flowchart of USB device authorisation taking provenance and context into of a newly inserted device into account

that a USB device can be plugged in at any time and make the operating system react, e.g. by loading the drivers for the device. This behaviour constitutes a potential security threat for two reasons. First, the number of drivers a malicious USB devices can pull is large and so is the resulting attack surface. And second, any USB device can be plugged into a machine while its user is not present, which potentially leaves no obvious trace of the device to the user after his or her return.

As a consequence, it can be argued that new (USB) devices should only be allowed if the user is actually present. An active lock screen, for example, might be a suitable indicator for the user not being present. Some devices, however, should probably always work. In particular, we might want to be able to activate new keyboards, because an existing keyboard might break and if the newly plugged in keyboard does not work, the user will be left locked out of the session. This problem might be solved by some logic such as "only one external keyboard shall be allowed". Furthermore, it is unclear what other external devices should have privileges similar to new keyboards. Accessibility devices, for example, might exist, and should be treated separately as well.

3.3.2 USB Devices Could Function Differently than Expected. As explained in Sect. 2.2, different device classes use the same connector type and so one device can masquerade as benign device to the user but actually identify as another device to the operating system. This can either be accomplished by re-programming the controller chip of a benign USB device or by altering the outer appearance of an USB device in order to let it appear as another device. In conjunction with the possibility of the USB infrastructure to invoke a de- and re-registration initiated by the USB device itself without the need of any user interaction such as a physical dis- and re-connection, the identification to the operating system can even be changed at any time while the USB device is plugged into a host. The device then gains different or additional capabilities, which the user did not expect.

3.3.3 USB Devices Should Just Work. Most of the design decisions of the USB specifications base on the requirement *Ease of Use*, which can be considered the most important impetus for the success of USB. In particular, those design decisions include the following according to Axelson [4]: One interface for many devices, hot pluggable, automatic configuration, and no user settings.

Those design decisions have led to the user's desire and expectation, that USB devices just work as soon as they have been plugged into their machine. This expectation makes it hard to design an User Experience (UX) that makes USB-borne attacks harder.

3.3.4 USB is simple by design. Arguably, USB has been designed to make it easy to implement. One manifestation is the lack of reliably identifying a USB device [27]. The structure of serial numbers used for device identification are not determined by the specification and such a serial number is provided by the device alone when connecting to a machine [10]. This means each time, a device connects, it can provide a different serial number. One exception is the specification of USB Type-C Authentication [38], which neither reached widespread adaption, nor effectively protects against USB-borne attacks (cf. Sect. 2.1).

In contrast, the interface connector Thunderbolt utilises cookies, which are created by the host and stored in a Thunderbolt device on first use [22]. This cookie can be provided by the device on subsequent connections in order to re-identify the device as already known device.

In the context of USB, such a concept of *Trust on First Use* cannot reliably be established, where a device receives a token when it is connected to a machine the first time and uses this token on subsequent connections to re-identify itself as already known and trusted device.

3.4 Data on USB Usage

Our proposed mitigation technique partially relies on some assumptions about typical real-life usage of USB devices in order to prevent or at least exacerbate atypical USB device behaviour. For example, when typically only one external keyboard is connected via USB, then it is reasonable to reduce the functionality of all additional keyboards and, e.g. filter special function keys. To substantiate

Table 1: Distribution of USB Devices by class

	≥ 1	≥ 2	≥ 3
Hub (I)	99.46%	94.02%	27.99%
Mouse	23.64%	0.27%	0.00%
Wireless	19.02%	2.45%	0.00%
Hub (E)	17.66%	2.17%	0.27%
Keyboard	13.86%	0.54%	0.00%
Camera	9.78%	0.00%	0.00%

these assumptions, we investigated the real-life usage of USB. To get hold of data about how people use USB we have crawled a popular Pastebin service for output of Linux’ `lsusb`, a tool for listing attached USB devices. We parsed the output and obtained a list of all devices plugged in to the machine. We discriminate internal and external USB hubs in order to get closer to the number of actual devices a user has plugged in. In addition to hubs, we identify several classes of devices which are of interested for our study, e.g. mice, keyboards, or cameras. We have matched devices based on the data available in that `lsusb` output which is vendor ID, product ID, and “Product String Descriptor”. We used services such as the “The USB ID Repository”[5] or “Linux Hardware”[6] to reconcile the data. Our data set includes 368 samples from 2017 to 2019.

Our findings can be summarised as follows: The median number of USB devices users have attached to their system is 2 with the lowest being 0 and the highest 14. We found that all but one reported a maximum of one keyboard. The only report of two keyboards was due to a Logitech “G11/G15 Keyboard” which has a separate USB device for “G keys”. We have accounted for wireless keyboards by way of looking for popular “unifying receivers” and Bluetooth receivers. In our data set, 19% (70 of 368) reports identified at least one receiver, 2.7% (10) more than one receiver. The highest number was three. We have manually investigated that report and identified one Bluetooth receiver and two Logitech unifying receivers. We guess that the machine had one receiver for a keyboard and one for a mouse. Our data set reports that 9.8% (36) have at least one camera attached. Unfortunately, the data does not show when the device has been attached, e.g. after boot or when the session was locked.

We note that our investigation of real-life USB usage has several limitations. First, there is no way we can assure that the users uploading that listing to pastebin did not manipulate the data. Second, it must be taken into account that the collected and analysed data is biased towards malfunctioning systems, because the pastebin service is rather used to exchange information when a system is not working properly. And third, the sample size is rather small, so our findings must not be understood as universal claims, but serve as strong indicators.

4 UNINTRUSIVE DEFENCE MECHANISM

The attack surface of contemporary operating systems is unnecessarily high. Using a simple set of rules, it is possible to fend of superficial attack attempts without involving the user. Existing defence mechanisms are either difficult to deploy or require user interaction. In fact, a whole line of research suggests to make the user decide whether a device should be authorised [14, 19]. Similarly, commercial software vendors will show a popup on every device the user attaches, e.g. G-Data[7] or Kaspersky[8].

The aim of this work is to show, how to increase the security as much as possible without changing the existing UX at all. We note that the result leads to a compromise in favour of usability. Our approach while developing the mechanism was to iteratively add as much security as is possible with the current UX before implementing changes to the UX in order to increase the security further. As the goal of this work is to gradually increase the cost of an attack, i.e. it aims for preventing the trivial attack of just plugging in a malicious device at any time. Once a protection mechanism against that trivial attack is active, e.g. by only allowing USB devices while the session is unlocked, we handle malicious devices inserted at a later stage or a device that re-inserts itself after a certain time.

The steps we took in our iterative process were the following:

(1) “Always Block” and “Never Block” USB policies
In this first case we plan to add an entry to the Control Center under the Privacy tab. This first step gives the user the option and responsibility to disable the ability to attach new devices. This step does not yet reach our goal of increasing the protection capabilities without changing the UX, because not only would the user have to manually activate or disable USB but also because then new devices might not work although the user expects them to.

(2) “Block only when locked” policy
The goal of this policy is to protect from attacks while the session is locked. By offering this policy we expect to increase the protection without the user noticing. Our argument is that the user is not present and hence cannot attach any devices. We would like to note that GNOME currently acts similarly when a mass storage device is inserted while the screen is locked: The filesystems contained on the device are not mounted in order to prevent attacks stemming from maliciously crafted filesystems. While conducting a small pre-study, we found a problem a user faces with a “Block only when locked” or “Always block” policy: If their existing keyboard ever turns out to be broken, they will be locked out of the session without a way to enter a password and get back in. To overcome this issue we handle input devices (esp. keyboards) differently from other USB devices. In particular, we check if the newly plugged in USB keyboard is the only input method available. If so, we allow it rather than blocking the device.

(3) Show new but blocked USB devices on the lockscreen
Our threat model allows for the user coming back to their computer after someone plugged in a malicious devices while the session was locked. We need to let the user know that they have a new device attached that is disabled and what the user can do to have the device activated. This requires a prominent widget in the lockscreen or after the user has unlocked their session which explains that a device has been blocked, because that

[5]https://usb-ids.gowdy.us/read/UD/
[6]https://linux-hardware.org/
[7]https://www.gdatasoftware.com/en-usb-keyboard-guard
[8]https://www.kaspersky.com/blog/badusb-solved/12539/

device could re-set itself after it determined that the user has returned. The concept of notifications on the lockscreen is well established not only in the GNOME desktop environment but also in mobile ecosystems for showing, e.g. new messages or media player controls.

(4) Handle new input devices naïvely
So far the security could be increased without changing the UX, e.g. the user did not notice the system behaving differently. Our approach is to gradually introduce security features and tighten them once they have been deployed. The goal of this step is to thwart the classical BadUSB attack, which involves a keyboard masquerading as another device, e.g. mass storage, with the methods the operating system already provides. To reach the goal, we lock the screen as soon as a new device with keyboard capabilities is attached. We argue that the user knows the concept of the lock screen already and that the notification about the new device will make the user suspicious if a surprising device is shown.

(5) Provenance at the keystroke-level
The goal of this step is to handle input devices more gracefully and not lock the user's session every time a new input device is attached as shown in fig. 1. Once we have exhausted the options for protection with the tools already available on the operating system, we can increase the protection profile by introducing new ways to deflect attacks. One way of achieving the set goal is to investigate the input device before either authorising the device or locking the screen. USB demands that an input device describes itself by way of a *report descriptor*. That descriptor identifies the keys an input device can press. We identify a set of keys which are dangerous and could allow keyboards which do not have the capability of pressing dangerous keys. Unfortunately, certain hardware exists which claims to be able to press all the keys of a commonly used keyboard, although intend to only press a limited subset. Such a hardware is a YubiKey which is used as a security token. The report descriptor includes all regular keys although they promise to press modhex keys only[9].
In this step we introduce the concept of tracking the origin of keystrokes and filter keys we deem dangerous. In particular, we do not allow external keyboards to press Ctrl, Alt, or Super, because those can be used to spawn a terminal and then launch an attack.

In summary, the decision making process for a newly attached USB device is based on context and provenance. Several conditions are checked before making the decision to either block or allow the new device (fig. 1).

Because we are aiming for reducing the impact on the UX and in particular not locking the user out of their session if they are unable to operate with their primary keyboard, we check whether the new device is the only currently available keyboard in the system. In that case an exception to the "block" protection level is made and the keyboard will be authorised.

[9]cf. http://web.archive.org/web/20171116184919/forum.yubico.com/viewtopic.php?f=6&t=96

5 IMPLEMENTATION

This section describes the modifications of the Linux environment we needed to make in order to develop our protection mechanism, although the concepts described in Sect. 4 can be applied to other operating systems as well. In fact, we have taken a first attempt of writing a prototypical implementation for Windows which we make available[10]. For our target environment, we needed to modify two sub-systems: First we needed to amend USBGuard to be able to integrate into a running session and then we needed to change the GNOME desktop to make use of the exposed functionality of USBGuard. All our modification have been published for inclusion into the relevant software packages and have either been included already or are in discussion for inclusion for the next release.

5.1 Backend with USBGuard

As an existing software component with protection capabilities we decided to build our mechanism around USBGuard. USBGuard is a daemon listening on udev events for new USB devices and then, if so configured, tells Linux to use a device. By using USBGuard we benefit from its implementation of the lower level kernel interaction, s.t. we did not need to re-implement that.

USBGuard uses two configuration files: `usbguard-daemon.conf` and `rules.conf`. The former holds a few variables like `InsertedDevicePolicy` and `ImplicitPolicyTarget`, while the latter is a file where it is possible to white- or blacklist single, or groups of, USB devices.

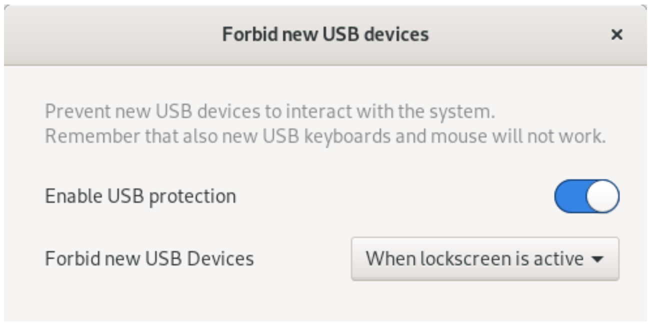
When a new USB device is detected, USBGuard performs a few sequential checks in order to decide whether a device should be enabled:

(1) From the `usbguard-daemon.conf` the value of `InsertedDevicePolicy` will be checked:
  - If it is `apply-policy` it will go to the step 2.
  - If it is `block` or `reject` the device will not be authorised or removed. No further checks will be performed.
(2) The `rules.conf` file will be evaluated:
  - If a rule matches the inserted USB device it will be applied. No further checks will be performed.
  - Otherwise it will go to the step 3.
(3) From the `usbguard-daemon.conf` the value of `ImplicitPolicyTarget` will be checked:
  - If it is `allow` the device will be authorised.
  - If it is `block` or `reject` the device will not be authorised or removed.

On top of that, from the CLI/DBus there is a function called `applyDevicePolicy` that can be used to override the decision that USBGuard took after the three steps listed above.

USBGuard configuration can be accessed and manipulated through DBus. In order to integrate it into the user's session, though, we needed to be aware of configuration changes. Alas, no signals were generated when a parameter was changed, making polling necessary for getting the potentially updated configuration values. Additionally, it was only possible to get and set the `InsertedDevicePolicy` parameter rather than `ImplicitPolicyTarget`. Without being able to modify that configuration variable, we could not

[10]https://github.com/ageinpee/USB-FilterDriver


Forbid new USB devices

Prevent new USB devices to interact with the system.
Remember that also new USB keyboards and mouse will not work.

Enable USB protection

Forbid new USB Devices: When lockscreen is active

Figure 2: GNOME Control Center USB protection entry

easily allow new USB device. We resolved these issues by contributing the necessary patches that are now merged and available in USBGuard[11].

5.2 Frontend with GNOME

Multiple GNOME components have been patched in order to insert the actual USB protection logic and to offer an usable and minimal UI. The components required for this project were:

- GNOME Control Center: we added an USB protection entry in the privacy tab. From there the users will be able to choose the desired protection level or completely disable it (fig. 2).
- Gsettings Desktop Schemas: an additional schemas has been added under `org.gnome.desktop.privacy` so that we can store the current protection level.
- GNOME Shell: we added a USB protection icon in the Shell top right indicator so that users have a visual confirmation that the protection is effectively active.
- GNOME Settings Daemon: this is were the actual protection logic lives. We created a new daemon that syncs the USBGuard configuration with the schemas on Gsettings and also it is the one that authorises new USB devices, using D-Bus.

We have published our modifications under a Free Software license and proposed them for inclusion to the project[12]. We expect the changes to be shipped with the upcoming major release of the GNOME desktop (version 3.34).

6 RELATED WORK

This section presents existing work that attempts to protect systems against rogue USB devices. Those attempts have in common that they empower the user to take the decision of whether to trust a device.

In 2007, the Linux kernel gained the capability of authorising a device before it is fully bound to a driver[13]. Since 2015, Linux can authorise USB interfaces separately [20]. Currently available security software packages for GNU/Linux to defend against attacks via USB use this mechanism to conditionally allow USB devices. The arguably most famous package is *USBGuard*[14]. It contains a daemon running in user-space watching for udev events which then prompts the user for authorisation of a device. It has an optional GUI which, interestingly enough, renders a subset of HTML when displaying the device name. So a malicious device can use markup like `<h1>`, `<b>`, or newlines (`0x0A`) to influence the rendering of the device on the screen. A different package is *Devdef* which prompts the user for consent to let Linux load a new *driver* rather than allowing a device [12]. This defence mechanism assumes that a malicious device intends to exploits vulnerable driver code rather than user expectation. In particular, once a driver has been loaded, e.g. by a benign device, a malicious could re-use that driver. Another approach that requires user interaction is USBCheckIn [14]: a physical box which the user needs to connect their input device to. The device then poses a quest which the user has to solve on the box itself. Similarly, SandUSB [21] requires the user to interact with a custom physical box in order to make a USB device work.

To the best of our knowledge, ProvUSB [35] is the most advanced description of a system to implement provenance-based security decision for data coming via USB. It takes the provenance of data on mass storage devices into account in order to provide a security solution for high security environments. Unfortunately, the release of the ProvUSB source code does not with a license which makes adoption harder. We extend ProvUSB's idea and apply it to other security relevant domains such as keyboard events or network configuration. Additionally, our focus is not high security environments but rather regular users' systems. And most importantly, our implementation has been released as Free Software which is about to be included in one of the most popular GNU/Linux desktops.

Our mechanism makes the security decision without the user. Such an automated system called USBlock has been proposed recently [25]. It applies a heuristic to detect "abnormal keypress sequences" and filters keystrokes based on their dynamics, i.e. the speed at which keys are being pressed. This will also block benign devices that are meant to type fast, such as a YubiKey which enters a user's password. In their work, they overcome this challenge by whitelisting the affected devices. While this works for devices known today, it does not scale for yet unknown devices. Our idea is similar in that we classify certain key strokes as more dangerous than others. Our idea is different, though, in that we take the provenance and context of the key strokes into account. Our approach does not require such a whitelist, unless the benign device needs to send the keys that we have classified as malicious. We appreciate that the problem has merely been shifted rather than fully solved. Nonetheless, we claim that our approach delivers a more usable system with higher protection capabilities and less false positives. Additionally, we provide source code with a Free Software license and have proposed it for inclusion into a major desktop system.

Other systems that attempt to defend against USB-borne threats are virtualisation-based and offer strong protection capabilities in expense for a changed UX, e.g. Qubes [28], Cinch [2], USBWall [19], or GoodUSB [34]. The latter comes with a modified Linux kernel and two user space daemons which decide whether a device is considered to be good [34]. For a new USB device it spawns a new virtual machine and observes how the VM reacts to the new USB device. It also prompts the user to let them decide whether to accept this new device. Another main reasons for preventing adoption is the unclear license. While the source code of their programs and modifications has been released, they are not available under a Free

[11]as commit `7cd1642a9f8b7cb0ca1ae15c8225912c329764b8`
[12]https://gitlab.gnome.org/GNOME/gnome-settings-daemon/merge_requests/75
[13]https://lwn.net/Articles/241980/
[14]https://usbguard.github.io/

Software license. USBFILTER [33] is system for inspecting packets on the USB and matching policies against the observed traffic. It could, for example, block SCSI write commands to pen drive to prevent (over)writing data on an external drive.

Other systems related to security of USB devices are: FirmUSB has a slightly different scope, namely to analyse firmware running on a USB device and checking whether it is benign [16]. Unfortunately, their software is not available as Free Software which hinders adoption.

The commercial market has come up with various "Endpoint Security" products[15] which aim to prevent the use of compromised USB devices that emulate keyboard behaviour. They generally display a popup when a new USB device has been connected and ask to user to enter, e.g. a PIN from that newly attached device. If the device does not have a physical keyboard (e.g. a YubiKey or a presentation clicker) it is possible to show an on-screen keyboard to manually enter the code with a mouse.

USB Type C authentication "has successfully pinpointed an urgent need to solve the USB security problem, its flaws render these goals unattainable" [36]. It will raise the bar for an attacker but since the firmware is not part of the authentication process, it cannot defend against certain classes of attacks. USB's success also draws from it's availability which is arguably stemming from cheap implementations. With USB Type C authentication, devices need to put a secure module which keeps keys secret which increases the price. In any case, users might still want to operate legacy devices without such a chip and they deserve protection.

7 DISCUSSION

We have presented how existing approaches are either difficult to deploy or comparatively hard to use. None of the protection mechanisms is in widespread use. Arguably, a minimally invasive method which requires as little user interaction as possible is easier to deploy and thus increases the general protection level. We have presented such a mechanism which takes context of the user's session and provenance of the potentially malicious USB event into account when making the security decision of allowing the device or its events. By making an automated decision the mechanism cannot afford false-positives, that is, rendering devices useless although the user expects it to work. To that end, we detect when the user is present, i.e. able to plug devices in and disable new USB devices otherwise. When the session is unlocked, we take the provenance of events from the USB device into account. Currently, we deny the second keyboard in a system to press keys we consider dangerous, e.g. Ctrl, Alt, Super. Once such a key is pressed, we inform the user about blocked keys and inform them about ways to escalate the privileges of the keyboard. We envision that once the system has been a built-in part of a major desktop environment, more and more provenance rules will be established. For example, a newly attached USB network card must not override the system's network configuration unless it appears to be non-functional. Or a webcam could only be allowed if the user has started an app that is able to interface with such a device. We believe that combining the context of the user's session, e.g. which apps are running or which network the user is connected to, and the provenance of system relevant events, e.g. overriding network configuration, allows for making many security relevant decisions without the user noticing.

Another strategy is to re-identify devices as best as we can, well knowing that the resulting additional protection is marginal, because it is not possible to reliably re-identify a device. But it does arguably increase the protection if the attacker needs to guess the serial number of the target device.

Because USB lacks a mechanism similar to Thunderbolt we cannot reliably re-identify a device. This makes defending against malicious devices more complicated.

We assume the legitimacy of our context and provenance rules to be backed by real-life usage data of USB. However, our data set has been collected from a highly biased source, i.e. we do not expect an average user to publish their data on Pastebin and the ones who do probably did so because of a malfunctioning machine.

We inferred that automated decision making without changing the UX increases the protection level. However, we have only conducted an early preliminary study and a more formal evaluation is needed to assert whether the goal of protecting users without them knowing could be reached.

[15] e.g. Kaspersky: https://www.kaspersky.com/blog/badusb-solved/12539/, Comodo: https://www.comodo.com/endpoint-protection/endpoint-security.php, or DeviceLock: https://www.devicelock.com/products/

REFERENCES

[1] Anne Adams and Martina Angela Sasse. 1999. Users Are Not the Enemy. *Commun. ACM* 42, 12 (Dec. 1999), 40–46. https://doi.org/10.1145/322796.322806

[2] Sebastian Angel, Riad S. Wahby, Max Howald, Joshua B. Leners, Michael Spilo, Zhen Sun, Andrew J. Blumberg, and Michael Walfish. 2016. Defending against Malicious Peripherals with Cinch. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, Texas, USA, 397–414. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel

[3] Apple, Hewlett-Packard Inc., Intel, Microsoft, Renesas Corporation, STMicroelectronics, and Texas Instruments. 2017. Universal Serial Bus 3.2 Specification. https://www.usb.org/document-library/usb-32-specification-released-september-22-2017-and-ecns

[4] Jan Axelson. 2009. *USB Complete: The Developer's Guide* (4th ed.). Lakeview Research, Madison WI.

[5] Gisela Susanne Bahr and William H. Allen. 2013. Rational Interfaces for Effective Security Software: Polite Interaction Guidelines for Secondary Tasks. In *Universal Access in Human-Computer Interaction. Design Methods, Tools, and Interaction Techniques for eInclusion (Lecture Notes in Computer Science)*, Constantine Stephanidis and Margherita Antona (Eds.). Springer Berlin Heidelberg, 165–174.

[6] G. Susanne Bahr and Richard A. Ford. 2011. How and Why Pop-Ups Don't Work: Pop-up Prompted Eye Movements, User Affect and Decision Making. *Computers in Human Behavior* 27, 2 (March 2011), 776–783. https://doi.org/10.1016/j.chb.2010.10.030

[7] D. Balfanz, G. Durfee, D. K. Smetters, and R. E. Grinter. 2004. In Search of Usable Security: Five Lessons from the Field. *IEEE Security Privacy* 2, 5 (Sept. 2004), 19–24. https://doi.org/10.1109/MSP.2004.71

[8] Silas Boyd-Wickizer and Nickolai Zeldovich. 2010. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1855840.1855849

[9] S. Butt, V. Ganapathy, M. M. Swift, and C. Chang. 2009. Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In *2009 Annual Computer Security Applications Conference*. 301–310. https://doi.org/10.1109/ACSAC.2009.35

[10] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. 2000. Universal Serial Bus Specification. https://www.usb.org/sites/default/files/usb_20_20181221.zip

[11] Chad Dougherty, Kirk Sayre, Robert Seacord, David Svoboda, and Kazuya Togashi. 2009. *Secure Design Patterns*. Technical Report CMU/SEI-2009-TR-010. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9115

[12] Moritz Duge. 2016. *Abwehr von BadUSB-Angriffen Mittels Kontrollierter Geräte-Aktivierung*. Bachelorthesis. HAW, Hamburg. http://edoc.sub.uni-hamburg.de/haw/volltexte/2016/3430/

[13] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2011. W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_

response/whitepapers/w32_stuxnet_dossier.pdf

[14] F. Griscioli, M. Pizzonia, and M. Sacchetti. 2016. USBCheckIn: Preventing BadUSB Attacks by Forcing Human-Device Interaction. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. 493–496. https://doi.org/10.1109/PST.2016.7907004

[15] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. 2009. Fault Isolation for Device Drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 33–42. https://doi.org/10.1109/DSN.2009.5270357

[16] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. 2017. FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2245–2262. https://doi.org/10.1145/3133956.3134050

[17] Moritz Jodeit and Martin Johns. 2010. USB Device Drivers: A Stepping Stone into Your Kernel. In *2010 European Conference on Computer Network Defense*. Berlin, Germany, 46–52. https://doi.org/10.1109/EC2ND.2010.16

[18] Samy Kamkar. 2016. PoisonTap - Exploiting Locked Computers over USB. https://samy.pl/poisontap/

[19] Myung Kang and Hossein Saiedian. 2017. USBWall: A Novel Security Mechanism to Protect against Maliciously Reprogrammed USB Devices. *Information Security Journal: A Global Perspective* 26, 4 (July 2017), 166–185. https://doi.org/10.1080/19393555.2017.1329461

[20] Stefan Koch. 2015. *Sicherheitsaspekte beim Anschluss von USB-Geräten*. Masterthesis. Universität Bayreuth, Bayreutch, Germany. https://epub.uni-bayreuth.de/3048/ Ursprüngliche Abgabe als Masterarbeit: 01. Juni 2015, Informationsstand dieser Überarbeitung: 25. November 2015, letzte Änderung vor Veröffentlichung: 23. Januar 2017.

[21] E. L. Loe, H. Hsiao, T. H. Kim, S. Lee, and S. Cheng. 2016. SandUSB: An Installation-Free Sandbox for USB Peripherals. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. 621–626. https://doi.org/10.1109/WF-IoT.2016.7845512

[22] Westerberg Mika. 2017. Thunderbolt Security Levels and NVM Firmware Upgrade. https://groups.google.com/forum/#!topic/linux.kernel/bAygpgSiKuA%5B1-25%5D

[23] Tobias Mueller. 2015. Framework for Fuzzing USB Stacks with Virtual Machines. In *INFORMATIK 2015*, Douglas W. Cunningham, Petra Hofstedt, Klaus Meer, and Ingo Schmitt (Eds.). Gesellschaft für Informatik e.V., Cottbus, 1901–1912. http://dl.gi.de/handle/20.500.12116/2176

[24] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. 2016. A Transparent Defense Against USB Eavesdropping Attacks. In *Proceedings of the 9th European Workshop on System Security (EuroSec '16)*. ACM, New York, NY, USA, 6:1–6:6. https://doi.org/10.1145/2905760.2905765

[25] Sebastian Neuner, Artemios G. Voyiatzis, Spiros Fotopoulos, Collin Mulliner, and Edgar R. Weippl. 2018. USBlock: Blocking USB-Based Keypress Injection Attacks. In *Data and Applications Security and Privacy XXXII (Lecture Notes in Computer Science)*, Florian Kerschbaum and Stefano Paraboschi (Eds.). Springer International Publishing, 278–295.

[26] Nir Nissim, Ran Yahalom, and Yuval Elovici. 2017. USB-Based Attacks. *Computers & Security* 70 (Sept. 2017), 675–688. https://doi.org/10.1016/j.cose.2017.08.002

[27] Karsten Nohl and Jakob Lell. 2014. BadUSB - On Accessories That Turn Evil. https://www.blackhat.com/us-14/briefings.html#badusb-on-accessories-that-turn-evil

[28] Qubes OS. 2018. Using and Managing USB Devices. https://www.qubes-os.org/doc/usb/

[29] Fahimeh Raja, Kirstie Hawkey, Pooya Jaferian, Konstantin Beznosov, and Kellogg S. Booth. 2010. It's Too Complicated, So I Turned It off'!: Expectations, Perceptions, and Misconceptions of Personal Firewalls. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration (SafeConfig '10)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/1866898.1866907

[30] Christian Seifert, Ian Welch, and Peter Komisarczuk. 2006. Effectiveness of Security By Admonition: A Case Study of Security Warnings in a Web Browser Setting. *In)secure Magazine* 1 (2006). https://doi.org/10.1.1.61.5696

[31] Jennifer Stoll, Craig S. Tashman, W. Keith Edwards, and Kyle Spafford. 2008. Sesame: Informing User Security Decisions with System Visualization. In *Proceeding of the Twenty-Sixth Annual CHI Conference on Human Factors in Computing Systems - CHI '08*. ACM Press, Florence, Italy, 1045. https://doi.org/10.1145/1357054.1357217

[32] Yang Su, Daniel Genkin, Damith Ranasinghe, and Yuval Yarom. 2017. USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 1145–1161. http://dl.acm.org/citation.cfm?id=3241189.3241279

[33] Dave Tian, Nolen Scaife, Adam Bates, Kevin R. B. Butler, and Patrick Traynor. 2016. Making USB Great Again with Usbfilter. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, Berkeley, CA, USA, 415–430. http://dl.acm.org/citation.cfm?id=3241094.3241127

[34] Dave Jing Tian, Adam Bates, and Kevin Butler. 2015. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/2818000.2818040

[35] Dave (Jing) Tian, Adam Bates, Kevin R.B. Butler, and Raju Rangaswami. 2016. 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)*. ACM, New York, NY, USA, 242–253. https://doi.org/10.1145/2976749.2978398

[36] J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. Butler. 2018. SoK: "Plug & Pray" Today – Understanding USB Insecurity in Versions 1 Through C. In *2018 IEEE Symposium on Security and Privacy (SP)*. 1032–1047. https://doi.org/10.1109/SP.2018.00037

[37] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. 2016. Users Really Do Plug in USB Drives They Find. In *2016 IEEE Symposium on Security and Privacy (SP)*. 306–319. https://doi.org/10.1109/SP.2016.26

[38] USB 3.0 Promoter Group. 2016. Universal Serial Bus Type-C Authentication Specification. Online. (March 2016). https://www.usb.org/document-library/usb-authentication-specification-rev-10-ecn-and-errata-through-january-7-2019 Revision 1.0.

[39] USB Implementers Forum. 2014. USB-IF Statement Regarding USB Security. http://web.archive.org/web/20160331174300/https://www.usb.org/press/USB-IF_Statement_on_USB_Security_FINAL.pdf

[40] Ka-Ping Yee. 2004. Aligning Security and Usability. *IEEE Security Privacy* 2, 5 (Sept. 2004), 48–55. https://doi.org/10.1109/MSP.2004.64

[41] Z. F. Zaaba, S. M. Furnell, and P. S. Dowland. 2014. A Study on Improving Security Warnings. In *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. 1–5. https://doi.org/10.1109/ICT4M.2014.7020633