A Transparent Defense Against USB Eavesdropping Attacks

Matthias Neugschwandtner
IBM Research – Zurich
eug@zurich.ibm.com

Anton Beitler
IBM Research – Zurich
bei@zurich.ibm.com

Anil Kurmus
IBM Research – Zurich
kur@zurich.ibm.com

ABSTRACT

Attacks that leverage USB as an attack vector are gaining popularity. While attention has so far focused on attacks that either exploit the host's USB stack or its unrestricted device privileges, it is not necessary to compromise the host to mount an attack over USB. This paper describes and implements a *USB sniffing* attack. In this attack a USB device passively eavesdrops on all communications from the host to other devices, without being situated on the physical path between the host and the victim device. To prevent this attack, we present UScramBle, a lightweight encryption solution which can be transparently used, with no setup or intervention from the user. Our prototype implementation of UScramBle for the Linux kernel imposes less than 15% performance overhead in the worst case.

CCS Concepts

•**Hardware** → **Buses and high-speed links;** •**Security and privacy** → *Embedded systems security; Systems security;*

1. INTRODUCTION

Attacks that leverage USB as an attack vector have evolved in the last years, although their potential has been pointed out a fairly long time ago [1, 15]. In the beginning, analogous to first DOS viruses on floppies, USB flash drives were merely used as a vehicle to deliver malware to potentially air-gapped devices, as was the case with Stuxnet [5]. Combined with an LNK exploit on shortcut files (CVE-2010-2568) in the case of Stuxnet, attaching the flash drive immediately launched the malware. Then efforts turned to USB itself, and in particular to the drivers necessary to support different USB devices. Because these drivers run in kernel mode, vulnerabilities in them can result in a full OS compromise [8, 9]. In contrast to malware delivery, this is a more advanced attack because it relies on the specifics of the USB stack and applies to all USB devices, instead of merely using USB flash drives as a delivery vehicle, which is comparable to malicious download links on web pages or e-mail attachments. Similarly, but on a different level, the increased availability of programmable USB devices that allow control over device specific parameters, can break the security assumptions of software that relies on those parameters through impersonation [11]. A well-known example is to mount an attack using a programmable USB device to act as a keyboard delivering keystrokes that spawn a command shell and executes pre-programmed commands[1]. Similar experiments have been conducted using smartphones [17]. Such attacks are also one of the motivations behind *BadUSB* [10]. With BadUSB, attackers that already control a host can leverage weak firmware update protections on some devices to re-program them to behave maliciously when plugged into another, victim host.

On the defense side, academic efforts have focused on limiting the privileges of USB devices on the host [16], while some manufacturers of flash drives, IronKey[2] and Kanguru[3] in particular, have come up with products that sign firmware to protect USB devices against malicious firmware modification.

However, the state of the art attacks and corresponding defenses have not yet covered the USB communications themselves. It is becoming increasingly clear that defenses are evolving to consider USB devices as hostile, and this means that attacks on the bus become relevant: we feel this situation is comparable to the early days of the Internet.

Indeed, there is no need to compromise the host to mount an attack over USB. Since data is transmitted in the clear, an attacker with physical access can eavesdrop on the communication. For example, sensitive data that is saved to a flash drive is available in cleartext on the bus, even if the flash drive itself encrypts the data at rest.

In this paper, we show how to intercept traffic on the bus from the host to the device, assuming only regular physical access to the USB bus, i.e. we do not cut wires but just need to plug a device into an arbitrary vacant port on the bus. At the same time, we present UScramBle, a USB transport encryption solution to defend against such eavesdropping attacks. UScramBle is completely transparent to the user and legacy devices that do not support it. We also present a performance evaluation of our implementation of UScramBle for the Linux kernel.

EUROSEC'16, April 18-21, 2016, London, United Kingdom

DOI: http://dx.doi.org/10.1145/2905760.2905765

[1] RubberDucky http://usbrubberducky.com/
[2] IronKey http://ironkey.com/
[3] Kanguru https://www.kanguru.com/

This paper makes the following contributions:

- We describe and implement a *USB sniffing* attack on USB 2.0 and earlier, where a device that is connected to a USB port eavesdrops on all (downstream) communications destined to other ports on the same bus. Unlike many previous USB attacks, this attack does not involve the host.
- We propose a clear threat model and a simple countermeasure that prevents this attack transparently (with no user involvement) and in a backwards-compatible manner: encrypting all downstream communications (transport encryption) with a key that is sent upstream with the first USB packet.
- We implement this countermeasure both on the host and device sides, and show that the performance impact is low.

2. BACKGROUND AND THREAT MODEL

2.1 Background

Before presenting possible USB eavesdropping scenarios as well as our proposed countermeasure, we first introduce relevant key aspects of the USB 2.0 Specification [3, Ch. 4.1.1].

USB is designed in a tiered star topology. Hubs are at the center of each star and facilitate bus extensions for additional hubs or peripheral devices. The first tier contains the host system comprising the Host Controller and the Root Hub. The Host Controller initiates all data transfers by polling the bus for transactions. Transactions start with a token packet describing the type and direction of the transaction as well as the USB device address and endpoint number. In a given transaction, data is transferred either from the host to a device (`OUT` transaction) or from a device to the host (`IN` transaction). The transaction source then sends a data packet or indicates it has no data to transfer. The destination acknowledges the transfer or indicates an error by means of a handshake packet. Logically, the host organizes data transfers in uni-directional pipes of different types, namely control, interrupt, bulk, and isochronous pipes. Devices provide endpoints for pipes, each device can have up to 32 endpoints. Endpoint 0 is reserved for the mandatory control pipe which serves for the exchange of configuration information during enumeration.

Devices are organized by the device descriptor, which foremost contains information about the device class as well as vendor and product to allow the host to select a suitable driver for the device. Further, a device features one or more logical configurations, which in turn consist of multiple logical interfaces that are associated with one or more endpoints.

The physical delivery of packets is facilitated by hubs. Hubs extend the physical lines by converting a single upstream port into multiple downstream ports. The upstream port of a hub connects it to the host while each of the downstream ports connects to another hub or peripheral device. The specification mandates a particular up- and downstream connectivity behavior for hubs as shown in Figure 1. While upstream connectivity is point-to-point, any downstream packet that is received on the upstream port is repeated at all enabled downstream ports, effectively broadcasting downstream traffic to all devices on the bus. Each packet is thus inspected by each device and further processed in case the destination address matches its own.

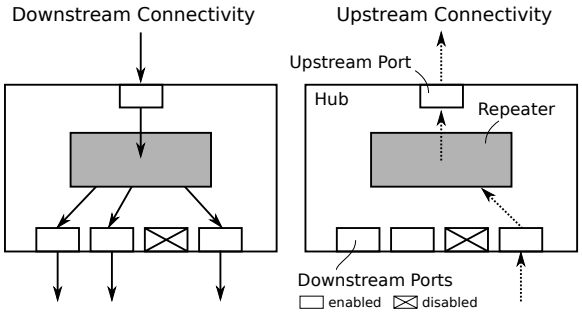

Figure 1: USB Hub Connectivity. Downstream traffic is broadcasted to all enabled downstream ports while upstream connectivity is point-to-point.

2.2 Threat model

We defend against an eavesdropping scenario where the attacker has physical access to the USB bus. While from a technical point of view the obvious option for eavesdropping might be a man-in-the-middle in-between the host and the device(s), we assume that the attacker, to be as stealthy and pervasive as possible, is limited to attaching as an ordinary device. Indeed, modifying an existing USB port or USB hub, or modifying a victim USB device, are much less interesting propositions to the attacker than simply inserting a malicious USB device on a seldom-used, well-hidden port.

Being an ordinary device on the bus, the attacker will see all downstream traffic, which is broadcasted to all devices in all tiers. However, the attacker will not have access to upstream traffic, which is seen only by hubs actually positioned in-between the sending device and the host.

Defending against a passive eavesdropping attack on the bus, we do not consider vulnerabilities in the software stack of the host or target device.

3. ATTACK

Following the technical description of a USB hub's internals as laid out in Section 2, we can summarize that hubs act as extensions of the bus topology and forward downstream packets from the upstream port to all downstream ports (Figure 1). This makes a sniffing attack on downstream traffic possible at any tier in a given topology.

3.1 Setup

To mount such an attack on a USB 2.0 bus one has to instrument a device to collect all packets that it receives irrespective of the destination address. However, with typical off-the-shelf USB device controllers this is not a trivial task since they implement most of the protocol in hardware. Configuration is done with the help of device, interface, control, and endpoint descriptors which specify all of the necessary parameters for enumeration and data transfer handling. What is left to the firmware is to manage the buffers associated with each of the configured endpoints. This architecture leaves little room to achieve the type of non-standard device behavior which is necessary for implementing a USB sniffer in a single low-cost peripheral.

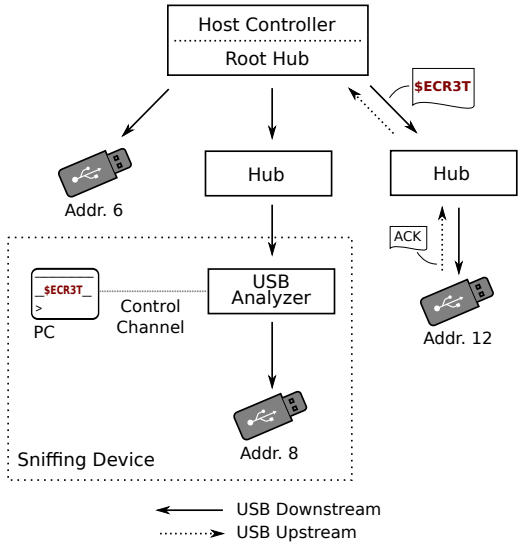

Figure 2: Attack Setup. The sniffing device can eavesdrop on all data that is sent from the host to the device with address 12, in spite of being connected to a different hub.

For our experiments, we chose to leverage dedicated USB analysis hardware. The setup of our testbed is shown in Figure 2. In this setup, two generic USB devices are attached at different tiers and enumerated with addresses 6 and 12, respectively. Our sniffer consists of a dummy device enumerated with address 8 and a hardware USB analyzer. The only purpose of the dummy device is to stay enumerated on the bus, thus keeping the corresponding downstream port at the upstream hub in an active state. This is needed because the analyzer's inspection link in itself is transparent to the topology and does not cause an upstream hub to enable the respective downstream port to the analyzer. The traffic seen on the inspection link is visualized by an analysis software on a PC which is attached to the analyzer via its control interface. In our experiments we use an Ellisys USB Explorer™ Model 200 analyzer and the Ellisys Visual USB Analysis software.

3.2 Attack Scenarios

A common USB use case is a scenario where potentially sensitive files are transferred from the host computer to a mass storage device. During this task a number of `OUT` transactions are issued by the host carrying the content of the files meant to be written to the device. The appropriate device acknowledges the transactions.

Consider a case where a file containing the string `$ECR3T` is written to the mass storage device enumerated with address 12, as it is indicated in the rightmost branch of Figure 2. The `OUT` transactions are broadcasted downstream by the host (solid arrows) and are thus picked up by the analyzer. A visual impression of the corresponding packet carrying the content of the file as shown by the Ellisys software is given in Figure 3.

This exemplary scenario demonstrates that USB eavesdropping attacks are not limited to certain device classes and apply to all application protocols. For example certain hardware-encrypted flash drives, while supposedly offering

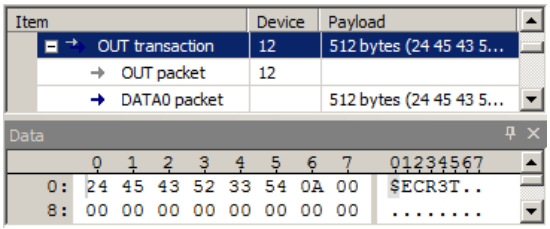

Figure 3: Analyzer screenshot with the `OUT` transaction carrying the content of a file being written to a mass storage device on the bus.

high levels of security for the stored data, expose the risk of eavesdropping the data while in transit from the host to the device. Similarly, a USB-to-serial conversion controller may receive security-relevant keystroke sequences from the user or may transport keys embedded into a firmware image meant to be flashed onto a controller. With today's small laptops featuring only a limited number of I/O ports and often even lacking onboard ethernet, USB docks which provide corresponding interfaces are common.

4. UScramBle

To defend against the USB sniffing attack, we propose to create a one-way confidential channel from the host to each device as part of the USB protocol. We now explain the design choices we made and how they mitigate the USB sniffing attack, and then detail implementation aspects of our prototype.

4.1 Design

From the threat model in Section 2, we know that the attacker can eavesdrop on all USB packets from the host to the device, but cannot eavesdrop on packets from the device to the host. In addition, the attacker is not in a position to alter packets, or spoof either the host or the device. To obtain a bi-directional confidential and integrity protected channel from this setting, we only need to add confidentiality to the downstream communication from the host to the device.

Therefore, unlike encryption on the Internet (e.g., TLS), where encryption protocols need to address confidentiality and integrity protection in both directions, as well as providing authentication of either or both parties, it is possible in this setting to use a much simpler and leaner design.

In particular, because the upstream communication (from the device to the host) cannot be eavesdropped by the attacker, this channel can be used by the device to send a key to the host securely, thereby achieving key exchange. Next, the USB protocol proceeds, with downstream packets being encrypted by the host with the shared key, and decrypted by the device upon reception.

From the cryptographic standpoint, our downstream eavesdropping threat model means that we need to use an encryption mode that is IND-CPA secure, i.e. ciphertexts are indistinguishable under chosen plaintext attacks (where the attacker has access to an encryption oracle). In particular, this means the encryption mode needs to be non-deterministic (encrypting the same USB payload results in a different ciphertext), therefore it needs to make use of nonces or sequence numbers when encrypting.

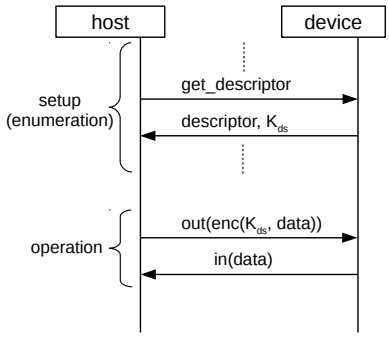

Figure 4: UScramBle protocol extension. The key is exchanged upstream during the device setup phase. The key is then used to encrypt downstream traffic for all following actual functional device operation.

In practice, we have chosen to keep track of the number of AES-block-sized (16 bytes) blocks transferred per-device as sequence number on both sides, and use AES-CTR with a 16-byte sequence number as initialization vector. We have chosen AES-CTR (which is IND-CPA secure under the assumption that AES is a pseudorandom permutation) because USB packets can be of any length, and CTR mode keeps packet lengths unchanged.

This defends against the USB sniffing attack in Section 3: the attacker merely sees packets that are encrypted with a symmetric key that the attacker cannot obtain.

We perform the key exchange very early in the setup phase of the USB device (Figure 4), which is called device enumeration. In this phase, right after the device has its address set, the host queries the device for its device descriptor. We chose to embed a 128 byte long key in the device descriptor's serial number field. The serial number is used by the host to distinguish multiple identical devices present on the bus. The serial number field in the device descriptor references an entry in a USB string table that contains the actual key. The key is randomly generated when the device is powered up.

After the key exchange succeeded, the host encrypts the payload of data packets that is sent to this device with the corresponding device key. Since they do not carry payload, we do not encrypt token and handshake packets. In case of control transfers, which are used to communicate all configuration information between host and device, data packets are also transmitted in plain. Note that configuration communication, such as the enumeration process, is separate from the device's functionality. For example, a mass storage device registers two bulk endpoints (`IN` and `OUT`) to handle storage access commands and data transfer. Both the storage access commands and the storage content are sent as data packets and thus encrypted by UScramBle.

UScramBle is opt-in on the host side: In case the device does not send a key, the host continues to communicate without encryption. While this might seem inconsequent at first sight, it is necessary in order to be compatible with devices that do not support encryption. If in turn, however, a UScramBle enabled device is attached to a non-encrypting host, it will deny communication. To the user, UScramBle is completely transparent since it does not require any interaction.

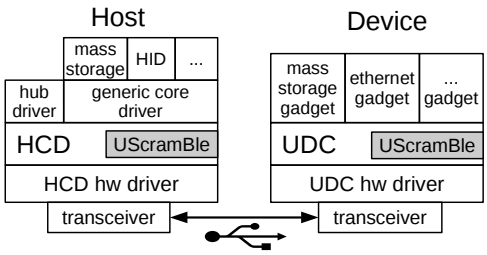

Figure 5: Host- and device side of the Linux kernel USB driver stack. UScramBle is implemented at the common device controller layer on both sides.

4.2 Implementation

We implement UScramBle as a modification to the Linux kernel version 4.3. Figure 5 shows how our prototype implementation of UScramBle fits into the USB driver stacks. Implementing it at the host controller driver (HCD) on the host side and the USB device controller (UDC) on the device side puts it in the sweet spot between various higher level functions and low-lever firmware that allows a layered design limiting code changes to a single place at each side of the driver stack. This makes the UScramBle implementation generic enough to be both device and function agnostic.

On the device side, the hardware specific UDC drivers are at the bottom of the USB driver stack. They process the USB request queue, which consists of `usb_request` structs. These structs describe an I/O request and are associated with an endpoint of the device. They contain a pointer to the transfer buffer, the length of the transfer buffer as well as a pointer to the I/O completion routine. The latter is called by the hardware specific UDC driver whenever an I/O request has finished. In the case of an `OUT` request, the completion routine is called after the data has been written to the transfer buffer and eventual DMA operations have completed.

On the top of the driver stack are the gadget drivers, which implement the device functionality. The Linux kernel features gadget drivers for various devices, such as mass storage, HID, Ethernet, serial or video devices. These gadget drivers interact almost directly with the hardware specific UDC drivers, except for a thin generic UDC layer that implements a common API. It is here where UScramBle intercepts the control flow before the gadget driver's completion routine is called and decrypts the contents of the transfer buffer. This way decryption is completely transparent to the gadget driver, which only sees the decrypted payload. We also implement the encryption key generation at the UDC layer, where we have access to the device configuration. The key is generated using the kernel's `get_random_bytes()` routine.

5. EVALUATION

We evaluated our implementation of UScramBle on a standard desktop host with Intel i7-6700 CPU, 32GB RAM, running stock Ubuntu 14.04 LTS except for our kernel modifications. On the device end, we use a USBarmory[4], an embedded device based on a NXP i.MX53 512MB RAM with a

[4]USBarmory https://inversepath.com/usbarmory

3.5K	Transfer Rate [KB/s]	95% Confidence Interval
Baseline	2816	±71.4
UScram-Ble	2781	±66.3
3.5K	**Request Time [ms]**	**95% Confidence Interval**
Baseline	6.34	±0.24
UScram-Ble	6.41	±0.22
10M	**Transfer Rate [KB/s]**	**95% Confidence Interval**
Baseline	17430	±6.05
UScram-Ble	17306	±6.65
10M	**Request Time [ms]**	**95% Confidence Interval**
Baseline	2938	±1.02
UScram-Ble	2959	±1.14

Table 1: Network transfer rates and request times measured using Apache benchmark.

USB device-side interface, running Debian Jessie, also with our kernel modifications.

We choose two real world scenarios for the performance evaluation, writing data to mass storage and transmitting network data via Ethernet. The corresponding functionality is implemented by the Ethernet and the mass storage gadget drivers of the Linux gadget driver framework. Each scenario is benchmarked with at least 30 runs, and confidence intervals are computed over these runs.

To benchmark the network scenario, we run lighttpd[5] on the device and fetch data via HTTP from the host using apache benchmark[6]. The results on the transfer rate and the time per request are given in Table 1. The overhead caused by UScramBle is 0.7% each for a file of 10MB and 1.2% (transfer rate) and 1.1% (time per request) for a file of 3.5KB, lighttpd's default page.

To benchmark the mass storage scenario, we use fio, the Linux flexible I/O tester[7]. We run it on the block device exposed by the gadget driver with a test file size of 100MB, direct I/O and a random write pattern to measure the throughput. We chose this worst-case workload because direct I/O and random access guarantee that we do not hit the filesystem cache, and, because encryption happens only on downstream communication, we only perform writes. These benchmark results are given in Table 2. The overhead caused by UScramBle is 15% in this workload. We have verified that the overhead is not caused by the encryption time itself. It is presumably caused by intercepting the DMA data path on the host driver stack, thus slowing down the transfer. Indeed, we checked that DMA transfers on the host occur in this mass storage scenario, but do not occur in the network device scenario.

6. DISCUSSION

The attack described in this paper does not fully apply to USB 3.0 and above. With USB 3.0, downstream broadcast of packets by the hubs has been removed in favor of packet routing. Hosts that are USB 3.0 capable keep track of which hubs the packet needs to pass to its destination and imprint this information into the route string embedded in the packet header. USB 3.0 hubs interpret the route string and forward the packet only based on the imprinted route. However, this packet routing mechanism only applies to the SuperSpeed lanes of USB 3.0. Any downstream traffic routed over the HighSpeed lanes is still broadcasted for reasons of downward compatibility. For example, a USB 3.0 mass storage device attached to a USB 2.0 hub is still susceptible to a sniffing attack. Also, given the vast number of USB 2.0 peripherals deployed, and some of the hubs even embedded in devices with a long lifespan such as monitors, we do think that the eavesdropping attack we describe will remain relevant for many years to come. In addition, because the USB 3.0 protocol has not been designed with a threat model comparable to the one in this paper, it is not excluded that the sniffing attack may be extended to USB 3.0 in the future.

	Throughput [KB/s]	95% Confidence Interval
Baseline	6877	±200
UScramBle	5851	±121

Table 2: Mass storage throughput as measured by the Linux flexible I/O tester.

A weak point of the solution is that the device is required to acquire sufficient entropy to generate the random cryptographic key. This is a well-known problem in practical applied cryptography, especially for embedded systems [7, 4]. However, a carefully implemented embedded system can generally have access to enough randomness to generate cryptographic keys, as was the case for our test platform.

7. RELATED WORK

Most research efforts regarding USB security are focused on protecting a victim host from a malicious USB device. Fuzz testing has shown to be effective in finding vulnerabilities in the host's USB driver stack [12]. GoodUSB [16] mitigates attacks that are based on exploiting the end user's expectation of a USB device's behavior. To this end, GoodUSB has the end user classify every device the first time it is connected. Based on the classification, the device is assigned a policy such as storage or cellphone and the OS kernel enforces this policy. Finer-grained policies, such as headset, which limit a HID's device functional capabilities require modification of the corresponding kernel driver. Along similar lines, approaches that are based on authentication mechanisms for USB [18, 19] help to protect the host against actively malicious devices by establishing a concept of trust between host and device. These solutions address a problem orthogonal to passive eavesdropping and thus do not protect against the sniffing attack described in this paper.

On the transport encryption side, Debaio et al. propose an elaborate three-factor security protocol [6] for mass storage devices attached via USB in a theoretical study. We believe a lean approach as presented in this paper takes into account the specifics of the USB protocol and is both sufficient and superior from a usability point of view, since there are neither passwords, nor an authentication server, which requires

[5]lighttpd https://www.lighttpd.net/

[6]Apache benchmark https://httpd.apache.org/docs/2.2/programs/ab.html

[7]fio https://github.com/axboe/fio

network connectivity, involved. Besides, UScramBle is device agnostic and not limited to mass storage devices.

Although far less popular and ubiquitous, Firewire is another peripheral bus that has been used for attacks in the past. Prominent attacks exploited Firewire's DMA capability to retrieve sensitive memory content such as cryptographic keys from the host [2]. Along the same lines, the newer Thunderbolt bus can be exploited [13]. DMA malware [14] is a generalization of this class of attacks, where the host is compromised through the DMA feature of a peripheral bus. While to the best of our knowledge both Firewire and Thunderbolt do not employ encryption and are thus susceptible to eavesdropping, both mitigation and attack would need to take into account the different network topology compared to USB.

8. CONCLUSION

In this paper, we present both an eavesdropping attack on USB communication as well as a protection against such an attack. The USB sniffing attack allows an adversary to passively listen in on all traffic that is sent from the host to a device. Our solution, UScramBle, effectively protects against such an attack by encrypting the traffic. The performance evaluation of our prototype implementation demonstrates that the overhead introduced by UScramBle is reasonable for real-world scenarios. Because UScramBle is completely transparent, i.e. it requires no user interaction, and compatible with legacy devices, it is effectively encrypting USB communication opportunistically, and successfully defending against USB sniffing attacks introduced in this paper.

9. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644412. The authors would like to thank Marc Stoecklin, Dhilung Kirat and Jiyong Jang for fruitful discussions on USB security.

10. REFERENCES

[1] D. Barrall and D. Dewey. Plug and Root, the USB Key to the Kingdom. BlackHat US, 2005.

[2] M. Becher, M. Dornseif, and C. Klein. Firewire – all your memory are belong to us. CanSecWest, 2014.

[3] Compaq, DEC, IBM, Intel, Microsoft, NEC and Nortel. Universal Serial Bus Revision 2.0 Specification. http://www.usb.org/developers/docs/usb20_docs/, 2005.

[4] J. Corbet. Random numbers for embedded devices. Linux Weekly News, https://lwn.net/Articles/507115/, 2012.

[5] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, 2011.

[6] D. He, N. Kumar, J.-H. Lee, and R. Sherratt. Enhanced three-factor security protocol for consumer usb mass storage devices. *Consumer Electronics, IEEE Transactions on*, 60(1), 2014.

[7] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium (USENIX SEC)*, 2012.

[8] M. Jodeit and M. Johns. Usb device drivers: A stepping stone into your kernel. In *European Conference on Computer Network Defense (EC2ND)*, 2010.

[9] J. Larimer. Beyond Autorun: Exploiting vulnerabilities with removable storage. BlackHat US, 2011.

[10] K. Nohl, S. Krissler, and J. Lell. Badusb – on accessories that turn evil. BlackHat US, 2014.

[11] G. Ose. Exploiting USB Devices with Arduino. BlackHat US, 2011.

[12] S. Schumilo, R. Spenneberg, and H. Schwartke. Dont trust your usb! how to find bugs in usb device drivers. BlackHat US, 2014.

[13] R. Sevinsky. Funderbolt – adventures in thunderbolt dma attacks. BlackHat US, 2013.

[14] P. Stewin and I. Bystrov. Understanding dma malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2013.

[15] A. Tetmeyer and H. Saiedian. Security threats and mitigating risk for usb devices. *Technology and Society Magazine, IEEE*, 29(4), 2010.

[16] D. Tian, A. Bates, and K. Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[17] Z. Wang and A. Stavrou. Exploiting smart-phone usb connectivity for fun and profit. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[18] Z. Wang and A. Stavrou. Usbsec: A defense to the ghost in your pocket. Kaspersky Security IT Conference, 2011.

[19] B. Yang, D. Feng, Y. Qin, Y. Zhang, and W. Wang. Tmsui: A trust management scheme of usb storage devices for industrial control systems. In *Conference on Information and Communications Security (ICICS)*, 2015.