The Impostor Among US(B): Off-Path Injection Attacks on USB Communications

Robert Dumitru
The University of Adelaide &
Defence Science and Technology Group
robert.dumitru@adelaide.edu.au

Daniel Genkin
Georgia Institute of Technology
genkin@gatech.edu

Andrew Wabnitz
Defence Science and Technology Group
andrew.wabnitz1@defence.gov.au

Yuval Yarom
The University of Adelaide
yval@cs.adelaide.edu.au

Abstract

USB is the most prevalent peripheral interface in modern computer systems and its inherent insecurities make it an appealing attack vector. A well-known limitation of USB is that traffic is not encrypted. This allows *on-path* adversaries to trivially perform man-in-the-middle attacks. *Off-path* attacks that compromise the confidentiality of communications have also been shown to be possible. However, so far no off-path attacks that breach USB communications integrity have been demonstrated.

In this work we show that the integrity of USB communications is not guaranteed even against off-path attackers. Specifically, we design and build malicious devices that, even when placed outside of the path between a victim device and the host, can inject data to that path. Using our developed injectors we can falsify the provenance of data input as interpreted by a host computer system. By injecting on behalf of trusted victim devices we can circumvent any software-based authorisation policy defences that computer systems employ against common USB attacks. We demonstrate two concrete attacks. The first injects keystrokes allowing an attacker to execute commands. The second demonstrates file-contents replacement including during system install from a USB disk. We test the attacks on 29 USB 2.0 and USB 3.x hubs and find 14 of them to be vulnerable.

1 Introduction

The Universal Serial Bus (USB) has become the de-facto standard for computer-peripheral connection. Since its original introduction in the late 90's, USB has replaced nearly all computer-peripheral connection standards. Simplicity, ease of use, and enabling low-cost implementation have always been prioritised throughout the development of the standard, propelling its popularity over the past two decades. Conversely, security has largely been overlooked throughout the development of USB. With the USB Implementers Forum arguing that "consumers should only grant trusted sources with access to their USB devices" [54], USB's security model relies on restricting physical access, rather than on tried and tested techniques, such as permissions, encryption, and authentication. In particular, operating systems typically immediately trust any USB device once connected, providing little feedback about the device's nature or capabilities. Given the ubiquity of USB, it is important to understand and characterise the attack surface and resultant threats presented by various usage and configuration scenarios. This characterisation can further enhance secure usage of USB, mitigating a wide range of USB-based exploits.

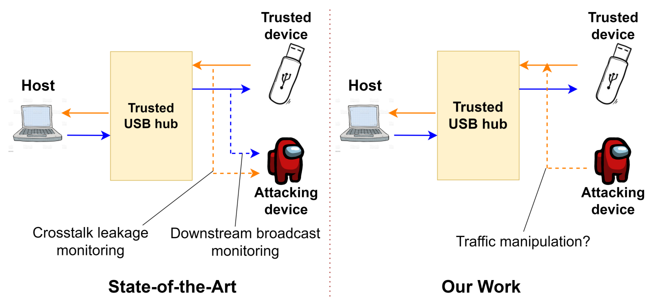

Figure 1: Off-path attacks on USB communications: (left) off-path traffic snooping by monitoring broadcasts [37] and using crosstalk leakage [44]; (right) we show an off-path attack that generates or manipulates the upstream traffic of other devices

With users often plugging untrusted USB devices into their computers [24, 51], numerous prior works have demonstrated attacks on the USB ecosystem via compromised devices. Attacks range from flash drives compromising hosts by pretending to be keyboards [31], to on-path entities such as hubs monitoring and manipulating USB traffic [30, 43]. Beyond on-path attacks by devices with direct data access, USB is also vulnerable to off-path attacks, where an attacker's device is not located on a direct path between the victim and the USB host. The left panel of Figure 1 summarises the state-of-the-art for off-path attacks on USB. In USB 1.x and USB 2.0, downstream traffic (host to devices) is broadcast on the bus making it observable for all devices on the bus traffic [37]. Su et al. [44] demonstrate that off-path attacks on confidentiality of upstream traffic are also possible, allowing devices to observe USB traffic sent by devices on adjacent USB ports due to electrical crosstalk.

Given the feasibility of off-path attacks on the confidentiality of USB data [44], in this paper we set out to investigate the feasibility of off-path attacks on USB data integrity. While this can be trivially achieved by on-path attackers (e.g., hubs) [30], we are not aware of any past demonstrations of an off-path attack that compromises the integrity of USB communication. Thus, in this work, we ask the following questions:

Does the USB protocol protect the integrity of upstream communication against a malicious off-path attacker? That is, can an untrusted malicious off-path device generate USB traffic that the host will attribute to a trusted device?

1.1 Our Contribution

In this work we show that USB does not protect the integrity of upstream communication even against off-path attackers. More specifically, we describe an *off-path USB injection* attack, which allows a malicious device located off the path between the victim device and host to send data which is accepted by the host as originating from the victim. See Figure 1 (right).

Attack Mechanism. Investigating the root cause of our attack, we find that when the host probes a device, the host and the hubs along the chain of connection fail to perform any verification that the response comes from the probed device. This allows a malicious off-path device to respond when the host probes a different victim device, resulting in the transmitted data being accepted by the host as having originated from the victim. We show how this mechanism can be used to send data to a host on behalf of devices that have been explicitly trusted by users through authorisation policies, thereby bypassing common USB defence strategies.

Attack Overview. In an attack scenario, an attacker-controlled device identifies as a benign USB device and performs the expected functionality for that device type. In addition, the attacker's device also monitors all downstream USB traffic. When it detects that the host probes the trusted victim device, it sends a malicious response that appears as if it were sent by the victim device. At the same time, the victim device will also respond to the probe, creating a race condition between the malicious and the victim devices. In the case that the malicious device wins the race, the host accepts the malicious response as if it were sent by the victim device.

Attack Implementation. We implement a USB 1.x malicious device that identifies as a mouse but sends malicious keyboard input when the host probes a separate victim keyboard device. We show that the attacker can consistently win the transmission race under vulnerable configurations, allowing a keystroke command injection attack. We further build a USB 2.0 device that identifies as a serial communications device and monitors communications of a USB flash drive, replacing the contents of files that will reside on the host when they are transferred from the drive. We show how this device can compromise a Linux installation.

Bypassing Protection Policies. Where a host's USB stack has been instrumented with a defensive device authorisation policy, our attack subverts this defense by falsifying provenance at the link layer. This in turn allows us to exploit any trusted device interfaces or communication channels. We show that our malicious devices bypass all defences that restrict device function based on such policies. We conclude with discussion on mitigation, recommendations for protecting existing systems and considerations for future system designs.

Summary of Contributions. In summary, in this paper we make the following contributions:

- We identify a new attack on USB that allows off-path malicious devices to inject traffic to the communications between a trusted victim device and the host. (Section 4)
- We design and build two injection platforms, one capable of targeting USB 1.x devices and the other targeting USB 2.0 devices. (Section 5)
- We investigate 29 USB 2.0 and USB 3.x hubs and find that 14 of them (48%) are vulnerable to at least one form of attack. (Section 6)
- We demonstrate how our attack can inject keyboard payloads and mass-storage device data, specifically hijacking data transfer to the host. (Sections 7.1 and 7.2)
- We show that the attack bypasses prior authorisation policy defences aimed at preventing USB attacks such as device masquerading. (Section 8)

1.2 Responsible Disclosure

Following the practice of responsible disclosure, we have shared our findings with the USB Implementers Forum (USB-IF), vendors of device authorisation software, vendors of vulnerable hubs, and the manufacturers of their internal hub chips in order to disclose our findings. At the time of writing there was no response from the USB-IF. A report detailing the findings was sent to those who responded. We note that the tested systems are all compliant with USB specifications, and the injection mechanism demonstrated in our work exploits a vulnerability in the protocol itself. See Table 4 in the appendix for the responses.

2 Background

First released in 1996, USB was designed to simplify the use of computer peripherals while replacing the plethora of then-common tailored interconnects with a single interface. The primary simplification USB introduced was that it allowed automatic self-configuration of peripherals upon plugging in, referred to as 'plug-and-play'. USB 1.x [10, 12] has two

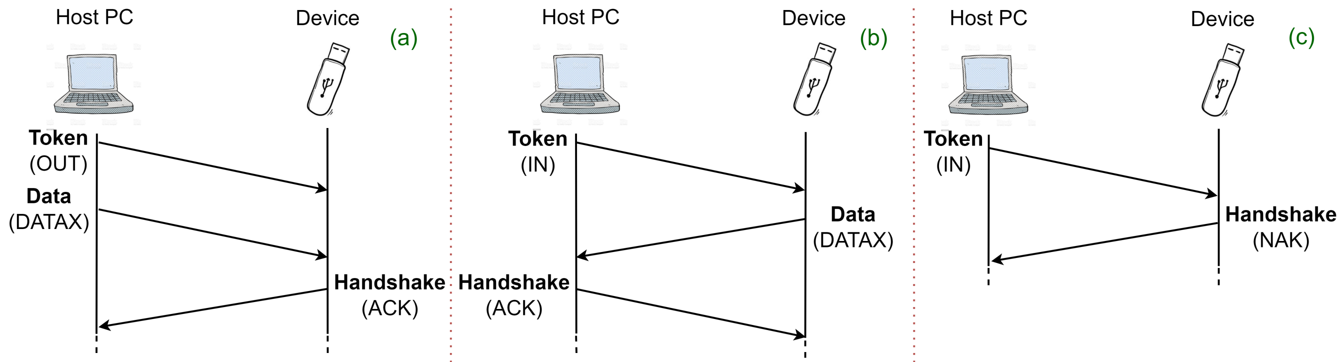

Figure 2: USB transaction examples: (a) host-to-device transaction, (b) device-to-host transaction, (c) device has nothing to send to host. The host first sends a token packet, identifying the target device within and the transfer direction. The host or the device then transmit data packets (with DATAX packet identifier). The handshake (or 'status') packet terminates the transaction.

data transfer modes which we refer to collectively as *classic-speed*: 1.5 Mbps Low-Speed (LS) and 12 Mbps Full-Speed (FS). Human Interface Devices (HIDs) and other devices undemanding of bandwidth continue to be made as USB 1.x.

USB 2.0. Later released in 2000, USB 2.0 [11] is an extension of the USB specification which is capable of operating at 480 Mbps in High-Speed (HS) mode. Due to its increased data transfer speeds, USB 2.0 was able to meet the needs of high-bandwidth applications such as imaging, data acquisition systems and mass storage devices.

USB 3 and 4. USB 3.x [3, 17, 18], initially released in 2008, is the latest major version of the protocol to have reached market maturity with speeds ranging from 5 Gbps (SuperSpeed) to 20 Gbps (SuperSpeed Plus). Backward compatibility with older device versions is built into USB 3.x host systems. Because of this and the extra cost of implementing a USB 3.x stack, devices only support USB 3.x if they stand to benefit from the increased bandwidth. In particular, the interface for HID devices is defined over USB 1.x [53]. Hence, these devices are unlikely to be implemented as USB 3.x. The most recent version, USB 4 [4], released in 2019 operates at speeds of up to 40 Gbps. As of this writing there are a few devices marketed as this version.

2.1 USB Communications

USB systems are structured in a tree topology. At the root is the host USB controller, which has an embedded *root hub* that provides a tier of attachment points for peripheral devices. Standard (non-root) hubs can be attached up to five in a chain to extend the number of USB ports, supporting up to 127 devices. We simply refer to standard hubs as hubs.

USB communication is host-arbitrated and non-encrypted. Downstream traffic originates from the host, which is broadcast in USB 1.x and 2.0, and unicast in later versions. Upstream traffic is unicast to the host in all versions of the USB protocol. The host manages the shared bus with poll-based Time-Division Multiplexing (TDM).

Endpoints. Endpoints are essentially data sinks and sources through which USB devices communicate, usually implemented as hardware buffers on the device side and as pipes in host-side software. All devices must support CONTROL endpoint 0, used for enumeration and status reporting. Devices can support up to 15 additional IN and OUT endpoints.

USB Transaction Protocol. All versions of USB use the same fundamental transaction protocol model. Data communication through USB happens in transactions, which consist of up to three packet transmissions comprising *token*, *data* and *handshake* phases. The host always initiates transactions by sending a token downstream, containing the intended recipient's address, a packet identifier defining transaction type, and the endpoint number. According to the USB standard, devices must only process and respond to tokens addressed to them while ignoring others. Data is only transferred in one direction during a transaction, and the direction is specified in the token packet identifier, with respect to the host. For example, the host will send an OUT token to indicate it will transmit data to a device during the data phase, see Figure 2(a). Similarly, hosts use IN tokens to probe devices for input to be provided in a data phase (c.f., Figure 2(b)). Otherwise, if devices have no data to send, they send a 'NAK' handshake (c.f., Figure 2(c)).

We note that no portion of the *data* or *handshake* phase packets identifies the source of the packet. Instead the source is implicit in which device is addressed by the *token* phase.

USB 2.0 High Speed Extension of Transaction Protocol. As described above, during OUT transactions the host controls the bus for the majority of time until the handshake phase, at which point the device either ACKs or NAKs the received data. This design can waste transmission time, for example in cases where the device is not prepared for data intake and thus NAKs the transaction. In order to mitigate this, USB 2.0 introduced an additional pre-transaction exchange before OUT communications at HS. Here, the hosts first send a PING token to query devices as to whether it is ready to receive data. The device can respond with a NYET (not yet) message or an

ACK, in which case the host begins the OUT transaction.

Backward Compatibility of USB 2.0 with USB 1.x. To reap the benefits of high bandwidth, in USB 2.0 systems all host–hub communication is delivered at HS (480Mbps), facilitated by Transaction Translator (TT) modules within USB 2.0 hubs. TTs perform downstream speed mode translation and buffer transmissions from classic-speed devices for repeating upstream at HS. Prior to transmitting LS or FS communications downstream at HS, a host will send a 'SPLIT' packet indicating to the hub that it must translate the next incoming transmission to a 1.x speed mode before passing it on to the recipient device. These SPLIT packets are sent prior to the token phase and only used in host–hub communication, making them not visible to end devices. SPLIT packets include an information field specifying the hub to which the end recipient device is connected, resulting in only that hub translating the subsequent transmission.

USB Routing. The design for backward compatibility discussed above introduces a form of routing-to-hub for the translated classic-speed traffic. Downstream classic-speed traffic is broadcast on the bus with a SPLIT header at HS, but then only translated at the target hub's TT for further broadcast at classic-speed. On top of this, hubs can be designed as single-TT systems (see Figure 3 top), where one TT handles all classic-speed traffic; or as multi-TT systems (see Figure 3 bottom), where each downstream port has its own TT. This also introduces routing to specific ports for classic-speed traffic through multi-TT hubs.

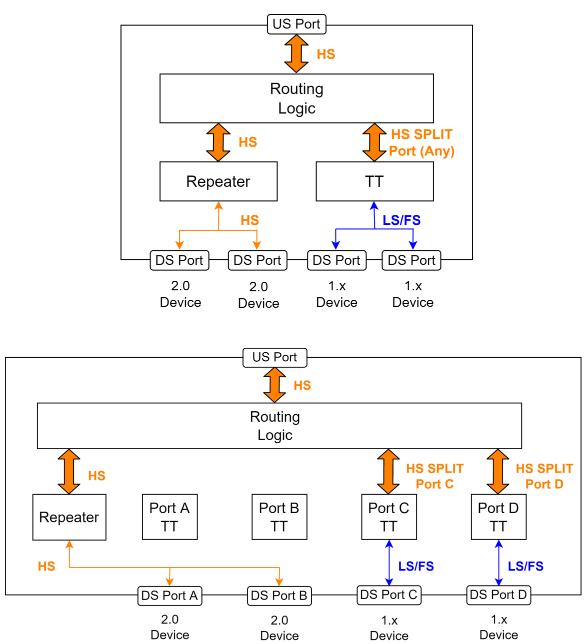

Figure 3: Traffic flows within single-TT (top) and multi-TT (bottom) hub with two (2.0) devices connected at HS and two (1.x) devices connected in classic-speed modes (LS or FS)

Enumeration. USB enumeration is the process of identifying a recently plugged in device and establishing a connection between it and the host. When a device is plugged in, the host will ask for its descriptor set containing self-reported (and not authenticated) information based on which the host can establish a connection. The host will then set the necessary output power to the device, its speed mode, and loading appropriate drivers. A newly connected device will use address 0 until the host assigns it a unique address early during enumeration.

2.2 Generic USB Device Architecture

Figure 4 shows the generic hardware architecture of a USB device. The PHY (physical layer transceiver) manages physical bus activity, allowing for sending and receiving the serial signal on differential data lines. The serial interface engine (SIE) module within the USB controller implements the transaction protocol, deals with time-critical operation and simplifies the microcontroller interface. The SIE handles token address checking before demultiplexing and facilitating information exchanges for the various endpoints.

2.3 Attacks on USB

The widespread adoption of USB among almost all PC, IoT, and embedded systems makes it an appealing avenue for exploitation. Notably, USB's lack of any access control mechanisms leads to simple and effective attack vectors, while requiring costly and complicated countermeasures.

We now review several categories of USB-based attacks and proposed protections, see [32, 33, 38, 40, 50] and references therein for more complete descriptions.

Obtaining Access. USB-based attacks rely on physical access to target systems and thus restriction of access to trusted devices is often used as justification for not securing USB systems. More specifically, the USB Implementers Forum (USB-IF) released a *Statement regarding USB security* [54], which says: "consumers should only grant trusted sources with access to their USB devices". However, typical USB-based attacks have minimal hardware requirements (size-wise) and malicious devices are often able to obscure their real behaviour from unsuspecting users. This results in users connecting devices from unknown origins to their systems, either inadvertently (e.g., supply chain compromise), naturally [51], or as a result of social engineering [24].

Attacks on USB Confidentiality and Integrity. USB's lack of any encryption and authentication mechanisms allows adversaries to eavesdrop on USB traffic, by both on-path and off-path entities. On-path or 'in-line' entities are those on the chain of connection between host and the subject device. Thus, attacks based on them require a stronger adversarial model than off-path attacks, as off-path entities do not require direct access to the targeted USB traffic.

On-Path Attacks. On-path entities can be passive devices such as protocol analysers acting as wiretaps. Alternatively,

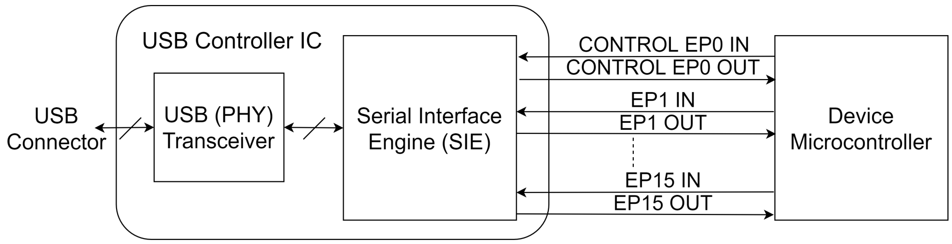

Figure 4: Generic hardware architecture of a USB device. The USB Controller IC consists of a PHY and SIE. The PHY manages the translation between physical analogue signals and the logical data they represent, the SIE implements the link layer transaction protocol, including address checking, and the microcontroller runs the application that transfers data via USB to/from the host.

they might also be active devices such as USB hubs which repeat communications passing through from one side to the other. Hardware key loggers [27, 29] are devices typically implemented as hubs with additional capability to record all keystroke traffic forwarded to a host, enabling capture of potentially sensitive information such as passwords. BadUSB 2.0 [30] is a hub which can compromise USB communications by performing full man-in-the-middle (MITM) attacks, including eavesdropping, modifying, replaying, fabricating, and even exfiltrating data sent between hosts and devices. Similarly, USBProxy [43] is an embedded system that uses USB On-The-Go [45][1] controllers to act as a MITM.

Off-Path Attacks. As all USB 1.x and 2.0 downstream traffic is broadcast on the bus, Neugschwandtner et al. [37] use a protocol analyser connected as an off-path device to directly monitor downstream communications to all connected devices. Such transmissions could include sensitive file contents in transit to a storage device, network adapter, or printer.

Similarly, upstream transmissions have been demonstrated to be observable by off-path devices [44] due to a 'crosstalk leakage' effect exhibited by the large majority of tested USB 2.0 hubs. By monitoring these leaked signals from adjacent USB ports, off-path devices can eavesdrop on unicast upstream transmissions of other devices to the USB host.

Electrical Attacks. Devices such as *USB Killer* [56] can permanently incapacitate host computers and other attached devices by discharging high voltage direct current over the USB data lines, damaging all connected circuitry. Robust electrical design protects against such attacks. This involves opto- or galvanic isolation of the USB port circuitry from the connected host controller or hub.

Device Masquerading. USB device controllers can also be programmed to emulate the operation of certain devices, capitalising on the lack of device authentication and USB's "plug-and-play" nature. These so called *masquerading attacks* use seemingly innocuous USB sticks with their firmware modified to emulate HID keyboards [7, 14, 26, 31]. Being able to feed arbitrary keystroke input enables adversaries to compromise computer systems in many different ways. Payloads of keystroke sequences can also be pre-loaded on-chip [31]. Alternatively, *URFUKED* [14] (the Universal RF USB Keyboard Emulation Device) incorporates RF communication for short range control over the keyboard emulator. Such attacks have become increasingly accessible with the advent of software-defined customisable USB devices [16, 31].

[1] Supplement to the USB specification that lets devices also assume the role of hosts, largely used in smartphones

2.4 Defences

As outlined above, USB's design has mostly overlooked security aspects, with the USB-IF leaving the inclusion of security measures to the discretion of system implementers [54]. This has resulted in a fragmented collection of bespoke defences [50], most of which are only enacted at one given layer within the USB stack. This is alarming, as Tian et al. [50] present many attacks which transcend multiple layers of the USB protocol stack, highlighting issues with non-holistic security solutions.

Policies. Device authorisation policies [1, 23, 35, 36, 39, 46, 49, 55] are a widely adopted protection against USB-based threats. These offer users varying degrees of control over device function. Protection policies vary from filtering communications to only allow certain devices, allowing certain interfaces within devices, and even only allowing certain interfaces to communicate with specific processes running on the host [36]. These policies also tend to fingerprint devices based on non-authenticated information that the device self-reports during enumeration. A survey conducted in 2019 found that over 40% of organisations apply protection policies [5].

Proxy Devices. Protective proxy devices [6, 15, 19, 21] can be placed between host and device in order to filter out unauthorised traffic. As these devices also provide physical separation between the attacker and the system's data lines, proxy devices also offer incidental protection against bus sniffing attacks. However, to be effective, proxy devices must be implemented in conjunction with complete system retrofits that physically disable other ports [1, 8, 28].

Cryptographic Protections. Cinch [2] introduces a cryptographic overlay protocol to augment USB with (bidirectional) encryption and authentication. While this approach protects against off-path confidentiality breaches, it also introduces more than 80% performance degradation due to throughput reduction. These types of defences incur additional burden as they require instrumentation of host USB stacks.

3 Threat Model

In our injection threat model there are at least two USB devices connected to a common host via some USB topology of USB hubs. We assume that one of the devices is our malicious injection platform, which is under the attacker's control. We further assume that the system contains a victim device, which is outside of the attacker's influence. This is the device whose communications the attacker would like to impersonate. Crucially, as per the tree topology of USB systems, the injection platform and the victim have logically separate, but physically shared, communication paths to the host. The system might have additional USB devices attached. These are considered bystanders, outside of the attacker's influence, and perform their function with unaffected USB communications.

Other System Components. We assume that all of the system components, except for the injection platform, are trustworthy. In particular, this includes the hubs on the path from the victim to the host as well as the host's operating system. We also assume that there are no malicious or compromised on-path entities which might help the injection platform to impersonate the victim.

Hardened Hosts. While not typically implemented by default on computer systems, we allow the host to employ a device authorisation policy in its USB software stack. Such policies limit the types of devices that the system supports and allows, e.g. via an authorisation list. Alternatively or additionally, the policy may require some form of user approval when a new device is plugged in [46]. The policy may restrict the nature of the communication with the device, e.g. to filter out malicious communication or to ensure that traffic is consistent with the communication protocol of the advertised device type.

Correct Implementation of Authorisation Tools. In case such an authorisation policy is present, we assume that these tools are correctly implemented and deployed, fingerprinting devices by any means at its disposal in the software stack. We also assume that such policy is correctly configured in that it accurately represents the user's intentions. We assume that the user and the policy trust the trusted victim device and allow its communication.

Policy Assumptions for Hardened Hosts. While the above assumes the correct functionality of authorisation tools, we do not assume any specific USB policy with respect to the injection platform, provided that it is physically connected to the host. The authorisation policy may even completely disallow communication with the injection platform, only allowing the USB communication with specific hand-picked trusted devices.

4 Attack Overview

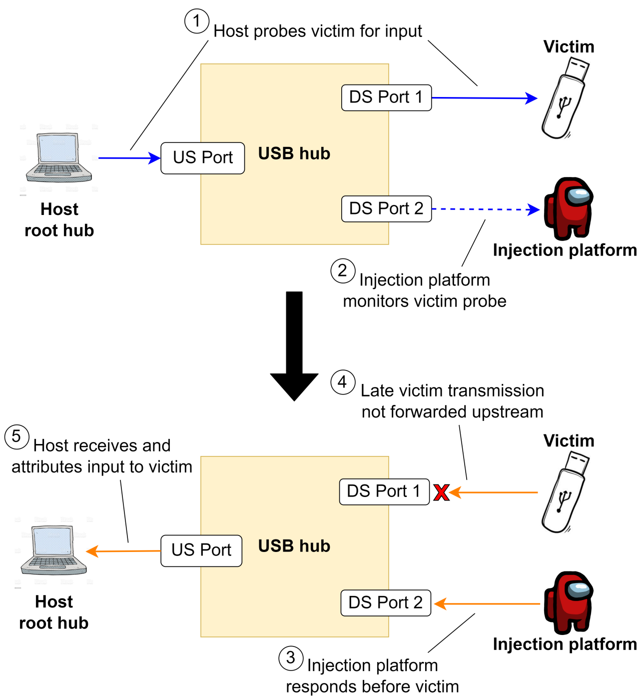

Figure 5: Injection arrangement and stages

Our injection platform displays two types of behaviours. It primarily functions as an innocuous USB device in its own right. Additionally, it inconspicuously injects upstream communications data to the bus, aiming to impersonate the victim. To that aim, we equip our injection platform with the ability to monitor the host's downstream communications for probes addressed to the victim, which trigger injections.

Injection. Figure 5 presents an overview of our injection attack. First, the host broadcasts a probe requesting input from the victim ①. The injection platform observes the host's probe ② and responds with an upstream data transmission ③ which matches the format of the expected victim response. Such behaviour is in violation of the USB specification. However, if the injection platform manages to respond before the victim device, the hub may accept the injected transmission and forward it upstream, while ignoring the victim's genuine response ④. Finally, we note that USB data and handshake responses do not carry address information. Thus, when a response arrives at the host, the host cannot distinguish between sources based on the received data, rather it attributes a response to the most recently probed device. Overall, in the case that the USB hub forwards the injected response upstream, the response from our injection platform is automatically attributed to the victim ⑤.

Bypassing Policies. Our attack exploits a vulnerability in the USB protocol brought about by a specification compliance assumption. Using our injection technique we can circumvent the USB authorisation policies on the host's hardware, as the host's USB controller cannot verify the source of the injected USB traffic. The crucial follow-on effect is that injection also bypasses software-based protection policies, as these implicitly trust the communications received from the host's USB controller to be correctly attributed.

Transmission Collisions on Hubs. Our attack exploits a race between the victim and the injection platform. If the injection platform manages to send a response before the victim starts sending its response, the attacker clearly wins the race and injects their response. However, injecting the whole response before the victim starts transmitting is quite unlikely, particularly when the attacker wishes to inject large amounts of data. When the transmissions of the victim and the injection platform collide, the hub needs to handle the collision. Figure 6 depicts such a DATA – DATA collision.

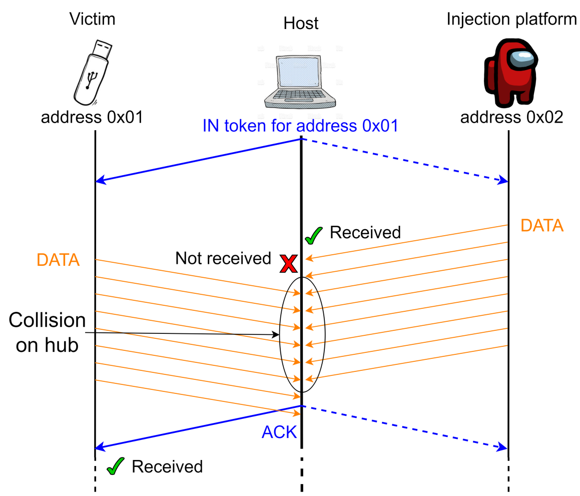

Figure 6: Sequence diagram of injection transmission and its collision with genuine victim transmission. Dotted lines represent transmissions that are observable by the injection platform despite it not being the intended recipient. Although the sequence diagram shows multiple arrows for DATA phases, these are part of singular continual transmissions only shown this way to represent the long transmission period.

Collision Resolution. In the case of a collision, the USB specification [11] permits two behaviours: a hub can treat the later transmissions as errors, completely ignoring them. Alternatively, the hub can detect the collision and send a 'garble' error message upstream to the host.

Hubs that exhibit the former behaviour, i.e. ignoring and discarding the later of incoming transmissions, are vulnerable to our injection attack. With collision-detecting hubs, injection can still effect a Denial-of-Service (DoS) against victim devices, blocking them from providing input.

5 Injection Platform Implementation

Having outlined the general principles behind our USB injection attack, in this section we describe the implementation of our injection platforms.

Triggering Injection. In USB 1.x and 2.0 systems, downstream communications are broadcast and can therefore be monitored directly by all devices in the USB topology, including off-path devices. Our injection platforms seek specific patterns in downstream traffic, which upon detection prompt them to inject traffic upstream. At a minimum, the final part of the expected pattern will consist of an IN token addressed to the victim device, which the host uses to probe it for input. As detailed in Section 4, the host will attribute received data according to whichever device was most recently probed.

Many classes of device implement their own communication protocol on top of the USB protocol, typically using multiple transactions and multiple endpoints. For example, hosts will request certain data from storage devices by issuing commands downstream using OUT endpoint transactions. To effect these higher level communications, our injection platform must recognise the relevant message sequence and subsequently trigger injections according to exchanges contextualised by the victim device class protocol, using packets crafted to match the corresponding format.

Existing Device Implementations. As described in Section 2, the USB transaction protocol is typically implemented by a dedicated SIE hardware module within a device's USB controller. Its function includes handling address checks of incoming tokens and the subsequent processing, i.e. when the token matches its device's address the SIE will write data to an OUT endpoint buffer or read data from an IN endpoint buffer. An existing device implementation can be turned into an injection platform through modification of its SIE, specifically the SIE's token address check function.

Target Device Type. To implement our injection platforms we modified hardware implementations of device SIEs within their USB controllers. This involved modifying the RTL source of cores implementing USB devices. Working at the hardware level offered high-fidelity control over timing which helped us ensure that our platforms win transmission races as required for the attack. As a possible alternative solution, there are some general purpose microcontroller families [34] with USB connectivity which implement the SIE in software/firmware and either support the USB interface directly (up to FS) or through an external PHY. While modifying the firmware of such an implementation could be a viable means of achieving our objectives, we have pursued hardware-based solutions instead because of the greater control (due to less abstraction) they offer over the platform function.

Hardware Setup. We have built two prototype injection platforms using existing implementations of USB device cores, which we deploy on FPGAs. The platforms are based on USB 1.x and 2.0 device implementations, one for each major version since the versions have different electrical interfaces and slightly different hardware behaviours. While they are largely similar, some properties of the devices differ between the implementations. We list these differences in Table 1.

	USB 1.x	**USB 2.0**
Original core name	FPGA-USB-V2 project [25]	USB2SOFT USB 2.0 device SIE [9]
Speed modes	LS (1.5Mbps) or FS (12Mbps)	HS (480Mbps)
Interface to USB	Direct to USB connector	12-pin UTMI+ to PHY Waveshare
PHY	Internal	USB3300
Adapted core from	Open source	Licensed IP
RTL source language	VHDL	VHDL
FPGA target	Xilinx Artix-7	Xilinx Kintex-7
Development board	Digilent Basys 3	Digilent Genesys 2

Table 1: Injection platform properties

5.1 USB 1.x Injection Platform

Core. The classic-speed (1.x) injector platform we have created is based on a modified (and debugged) core [25] written in VHDL, which is freely available for distribution.[2] The core contains elements that perform the functions of all hardware modules within a device controller as in Figure 4 (PHY and SIE), however its structure is monolithic rather than partitioned into those distinct modules. In conventional devices, communication across all endpoints would be handled by the device microcontroller. This core instead defines functionality for device enumeration, performed by communication over control endpoint (0), directly in hardware. The core also allows for communication over input/output (non-control) endpoints to be implemented directly in hardware. The core interfaces directly with the USB differential data lines D+ and D- and can be set to operate as either a LS or FS device.

Deployment. We instantiated the core on a Xilinx Artix-7 FPGA housed on a Digilent Basys 3 development board. We directly connect the USB data lines from a spliced USB cable to 3.3V general-purpose I/O pins on the board. The device is shown in Figure 11 in the appendix.

To configure the device to work at LS, we pull up the D- line to 3.3V across a 1.5kΩ resistor, this signals the speed mode to the host and sets the associated differential signalling polarity. To work at FS, the D+ line must be pulled up instead.

5.2 USB 2.0 Injection Platform

Core. The HS injection platform is based on an adapted licensed device core IP [9] written in VHDL. This core instantiates a SIE and implements enumeration functionality, handling all CONTROL interactions over endpoint 0. FIFO interfaces support IN and OUT endpoints.

The core connects to an external PHY across the low pin count interface – UTMI+ (ULPI) [41]. This is the de facto standard interface for SIEs interacting with USB transceivers. We use the interface with an 8-bit wide data bus and control signals which are clocked by the PHY with a 60MHz signal derived from transitions on the 480Mbps HS data lines.

Deployment. We have ported the core to a Xilinx Kintex-7 FPGA housed on a Digilent Genesys 2 development board. The board is connected through its general-purpose I/O pins, which are capable of switching at high frequency, to the UTMI+ pins of a Waveshare USB3300 PHY board. Figure 12 in the appendix shows this device.

5.3 SIE Modifications

As outlined in Section 2.1, Endpoint 1 is typically the main input (IN) endpoint used by devices and endpoint 0 is the CONTROL endpoint used for conveying setup information during enumeration. Since we do not wish to interfere with our victim's enumeration, we configure our platforms to only inject endpoint 1 traffic.

As an example, in the USB 1.x device implementation RTL source we identify the following line which defines the device's address check behavioural logic on incoming tokens:

```
if (token_ad /= new_usb_addr) or
  (pid /= not token_in(16 downto 13) then
```

where '/=' is the VHDL inequality operator. When this `if` statement evaluates as true due to an address mismatch (first condition) or transmission error (second condition), the currently inspected token packet is no longer processed and the device waits for the next token. We transform the device into an injection platform by modifying this line to the following:

```
if ((token_ad /= new_usb_addr)
  and (endp /= "0001")) or
  (pid /= not token_in(16 downto 13) then
```

Now our device can process incoming tokens with address mismatches if they are intended for endpoint 1. We note that with this modification the platform would inject endpoint 1 traffic on behalf of itself and victim devices connected on the bus in the same speed mode. We thus further alter the IN endpoint 1 behavioural logic to only send data for probes with address mismatches, allowing the device to ignore its own traffic, sending NAKs to all of its own probes. We similarly modify the USB 2.0 device implementation to the same effect.

6 Testing USB Hubs

In this section we evaluate the hardware setups that are vulnerable to our injection attack. We evaluate a total of 29 hubs and

[2] The injection platform bitstream and source RTL code are available at: https://github.com/0xADE1A1DE/USB-Injection

find that 14 of them are vulnerable. We then investigate features that correlate with some of these vulnerabilities. Finally, we test the impact of bus topology on our injection attack.

6.1 Testing Methodology

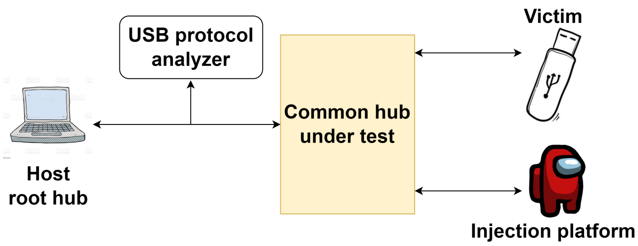

Figure 7: The testing environment.

Topology. To test for injection attacks, we configure our injection platform to inject a unique and easily identifiable data sequence into the USB communication stream. We then set up an experiment in which both an injection platform and victim device operating at the same speed mode are connected to the host PC through the hub under test. See Figure 7. We refer to this hub as the *common hub*.

Experimental Setup. Between the host and the common hub we attach a Totalphase Beagle USB 5000 protocol analyser, which observes and logs all traffic on the link between the common hub and the host. Because both the victim device and the injection platform are connected to the host via the common hub, the protocol analyser also captures all traffic between the host and these devices. We repeat each experiment three times, once for each of the operating speeds. For the LS and FS operating speed, we use our USB 1.x injection platform from Section 5.1, with keyboards as the victim devices. For the HS operating speed we use our USB 2.0 injection platform from Section 5.2 with mass storage victim devices. The specific models are listed in Table 2 in the appendix.

Communication Analysis. When our injection platforms inject transmissions during a victim's time slot the genuine victim response is also sent, causing a collision at the common hub. A vulnerable hub that enables injection continues to forward the first incoming transmission upstream and blocks all subsequently arriving simultaneous transmissions. In our experiments this result is evident from observing that the protocol analyser log attributes the unique data sequence that the injection platform transmits to the victim device's assigned address. Otherwise, if the hub sends a garble/error sequence upstream when it detects a collision, the unique data sequence does not appear in the protocol analyser's log.

6.2 Targeting USB 2.0 Hubs

We begin our investigation with a test of 16 USB 2.0 hubs. These include thirteen standalone hubs, which are attached via a USB cable to a host, and three embedded hubs that motherboard vendors embedded in their products to increase the number of supported USB ports. Table 3 in the appendix lists all hub models tested and their advertised IDs. Note, the embedded hubs were tested by carrying out the end-to-end attacks as described in Section 7.

Injection Results. We find that 13 out of the 16 tested hub devices are vulnerable to some form of injection attack. One device is vulnerable to both USB 1.x and USB 2.0 injection, seven devices only to USB 1.x injection and four devices only to USB 2.0 injection. Interestingly, all of the embedded hubs we tested are vulnerable to injection and attacking motherboards that feature them is possible even without any external USB hubs.

Anomalous Hub Behavior. Three of the hubs we tested exhibit specific behaviour not observed in other hubs. Two unlabelled hubs, marked as USB 2.0 hubs, only operate at USB 1.x speeds. We were unable to find similar hubs and do not know if the issue affects all hubs of the same models or only the specific ones we tested. The other hub displaying specific behaviour is the embedded hub in the Micro-Star PRO Z690-A motherboard (marked with (✓) in Table 3). The motherboard has only two exposed ports that seem to show an unbalanced behaviour. Injection from one of the ports works consistently. However injection from the other is intermittent, with the victim device sometimes winning the race, preventing the injection. A possible explanation for this unique behaviour is that the hub uses asymmetric downstream port arbitration or switching.

DoS Results. Finally, we tested the hubs with a DoS attack, in which the injection platform transmits a NAK in response to every probe that the host sends to the victim. In vulnerable hubs, the injection platform wins the race and the host accepts the injected NAK. In non-vulnerable hubs, the hub detects the collision and sends an error signal, effectively deleting the victim's response. Thus, in either case, the denial of service attack is effective. The only exceptions are the embedded hub described above, where in one configuration the victim sometimes wins the race, overcoming the attack; and multi-TT hubs against 1.x traffic injection, since the downstream probes do not reach the injection platform.

6.3 Targeting USB 3.x Hubs

We now turn our attention to USB 3.x hubs. We tested 13 such hubs, also summarised in Table 3 in the appendix. For backwards compatibility, USB 3.x hubs consist of two logical hubs, one handling USB 3.x SuperSpeed traffic, while the other handles compatibility with USB 2.0 devices. As our injection platforms do not operate at SuperSpeed, the experiments only test the internal USB 2.0 hubs. Interestingly, when enumerated, the internal USB 2.0 hubs identify themselves as USB 2.1 hubs. The USB specification [17] does not specify a version 2.1. We suspect that this version number is used to differentiate internal hubs from pure USB 2.0 hubs.

Overall, we find that USB 3.x hubs are less vulnerable to injection attacks, with only one of the 13 we tested allowing injection. USB 3.x hubs are still vulnerable to denial of service attacks for USB 2.0 victims. USB 1.x victims can also be attacked with USB 1.x denial of service, but only in the case that the internal hub they connect to is single-TT.

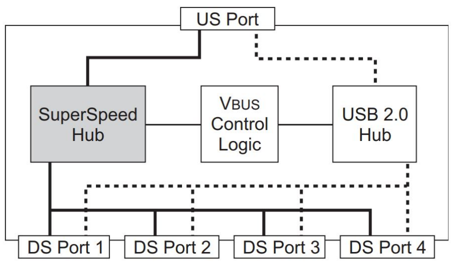

Figure 8: USB 3.0 Hub Architecture [17]

6.4 Root Hubs

Our investigation so far focused on standard hubs that are used for the internal levels of the USB network tree. *Root hubs* that provide the first tier of attachment points to a host system and are structured differently to standard USB hubs. Their architecture and operational model is defined in the eXtensible Host Controller Interface for USB (xHCI) specification [22], which standardises methods for communication between USB software and hardware. xHCI specifies mechanisms for maintaining port association with connected devices that ultimately routes transmissions to those devices, thus there are no broadcasts of downstream USB 1.x and 2.0 traffic across root hub ports.

As our injection platforms relies on broadcast traffic to time the injection, we cannot expect the attack to work when downstream traffic is not broadcast. Indeed, we have tested injection using both platforms against multiple xHCI root hubs and found none to be vulnerable because the injectors have no visibility of probes sent to victim devices.

6.5 The Impact of Transaction Translators

Observing the results above, we note that the behaviour depends on the configuration of transaction translators in the hub. Specifically, we find that multi-TT hubs are not vulnerable to USB 1.x injection, because the injection platform does not observe the host's probes.

When performing USB 1.x attacks against single-TT hubs that resist the attack, we observe that the hub sends a 'SPLIT-ERR' (error) message upon attempted injection. We hypothesise that the hub detects the collision and sends the error message instead of a garbled signal.

6.6 Exploring USB Topologies

In the testing described so far the victim device and the injection platform have been connected directly to the common hub. We now investigate the effects that introducing additional tiers of hubs between victim and injection platform has on injection. USB allows up to 7 hub tiers, where the root hub is the first tier, so up to 5 chained hubs can be cascaded from the root. In all the experiments, we use a vulnerable hub as the common hub and safe hubs otherwise.

USB 2.0 HS Injection. We find that is it possible to inject HS traffic in all cases where the injection platform is connected at a hub tier that is not further away from the host than the victim. When the injection platform is further than the victim, injection does not work. We believe the reason to be that the time taken for the HS signal to propagate through a hub repeater is significantly greater than the time by which our injection platform undercuts the victim's response.

USB 1.x LS/FS Injection. When the common hub is a USB 2.0 or 2.1 hub, we find that injection of USB 1.x traffic only works when both the victim and the injection platform are attached to it directly. This is because downstream USB 1.x traffic is delivered at USB 2.0 HS (immediately following a SPLIT message) and is only translated to a USB 1.x speed at the hub to which the receiving device is attached. Consequently, if the victim device and the injection platform (operating as a USB 1.x device) are not connected to the same hub, the injection platform cannot observe broadcasts of translated downstream traffic sent to the victim device.

However, we note that when injecting against a USB 1.x victim, a USB 2.0 injection platform can target the hub the device connects to instead of targeting the device itself. We verified that, by injecting HS traffic with our USB 2.0 injection platform on behalf of the hub, the injection platform can spoof the hub's translated USB 1.x traffic, thereby confusing the host to accept the injected traffic as if it originated at the USB 1.x victim device. Therefore, we can inject USB 1.x traffic from the same hub tier only when connected via USB 1.x on the same hub, or we can inject translated HS traffic with a USB 2.0 injection platform connected at a closer hub tier than the victim. Note that this includes placing the injection platform at the same tier as the hub the victim connects through.

The result is different when using a USB 1.x hub operating at LS/FS speeds as the common hub. In this case, all hubs connected on the downstream of the common hub revert to operating as USB 1.x hubs. Consequently, no traffic translation occurs, and the attack works irrespective of the number of intermediate hubs between the common hub and the victim device or the injection platform.

7 Injection Attacks

In previous sections we demonstrated injection of USB traffic. We now investigate the security impact of such injection. Specifically, we demonstrate two attacks, one injecting commands through keystroke data and the other compromising system security by injecting file system contents.

7.1 Keystroke Command Injection

Keyboard USB Stack. HID Keyboards typically operate at LS and use endpoint 1 as their main and only input endpoint. They are simple devices which report character key press and release events. As such, beyond the USB transaction protocol there is no higher-level protocol used by hosts to elicit data. Thus, we have directly adapted our USB 1.x injector to demonstrate injection of keystroke commands to the host, like what might be sent in a protocol masquerading attack.

Attack Payload. In our ad hoc microprocessor application implementation we program a payload of data packets into the platform core directly in hardware and tie their provision to press events for buttons on the board. The payload sequence opens a Command Prompt on a Windows system.

DoS-Switch. We further instrument the platform with a 'DoS-switch' enabling selective injection of NAKs on behalf of the victim to block its inputs from being forwarded. Blocking can be useful under circumstances where the adversary performing injection wants to inject an uninterrupted payload sequence of packets. The NAKs are sent only when the injector is not providing DATA packets of its own.

Experimental Setup. We configure our injector to identify as a HID mouse operating at LS. We connect both the victim and our injection platform to a host through a common hub (one previously found to be vulnerable), again placing the protocol analyser on the hub's upstream connection.

Results. We successfully perform keystroke injection attacks against keyboard victims. We open a Windows Command Prompt, and using the protocol analyser we observe that the injected traffic is attributed to the victim keyboard's assigned address. Unplugging the keyboard and pressing the same buttons on our connected injection platform results in no key presses, further confirming that injections have taken place and it has not somehow mistakenly fed keystrokes as a mouse. We verify the function of our DoS-switch by pressing keys on the victim observing that with the attack enabled, keystrokes do not pass through.

Latency. Being able to inject successfully means our attack platform can undercut the victim keyboard probe responses. We confirmed this by inspecting the respective devices' packet timings in the protocol analyser traffic capture. We do not expect other keyboards from different manufacturers to have response times fast enough to pose a concern. The platform should consistently win transmission races and no further modification is needed to speed up response times.

Attacking Gaming Keyboards at FS. We alter the USB 1.x injector to work at FS and find that it can also successfully inject against a low latency gaming keyboard at FS with a 1 kHz poll rate. This is because gaming keyboard latency is bounded by the high polling rate and not outright hardware response times.

7.2 Hijacking File Transfers

For our second use case we adapted the USB 2.0 injection platform for compromising the communications of HS flash drive victims. Using it we can hijack device-to-host file transfers. The platform listens for data requests sent to a flash drive victim, which stimulates it to inject data that ultimately alters the contents of files that end up resident on the host.

We further demonstrate this capability in a use case where we compromise a Kali Linux OS image in a boot from USB.

Mass Storage Device Stack. Here we describe aspects of the mass storage device (MSD) class relevant to the use case.

Endpoint 1 is typically the main data input endpoint for MSDs. This means we can retain the modifications that made the original HS injection platform. However, injection has increased complexity in this use case because MSDs implement a *Command/Data/Status* transport protocol across multiple USB transactions and over multiple endpoints. Its operation is largely analogous to USB's transaction protocol. This protocol uses the Small Computer System Interface (SCSI) [42] command format. For our purposes we are not interested in the information transfers that establish a link between the host and MSD file systems, rather we are only specifically interested in the exchanges that take place during a file transfer on established links. Once a link has been established, the host periodically sends the SCSI `Test Unit Ready` (TUR) *Command* which essentially acts as a keep-alive message. This is followed up by the device sending a *Status* message to the host indicating that it is ready for another command.

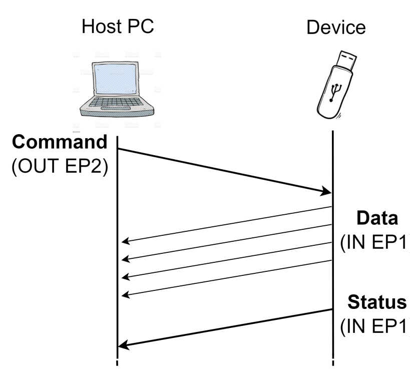

Figure 9: *Command/Data/Status* sequence in a device-to-host mass storage file transfer. Each arrow in the diagram is representative of an entire 3-stage USB transaction.

When the host wants to initiate a device-to-host file transfer it will issue a SCSI `read(10)` *Command* requesting transfer by communicating through OUT endpoint 2, illustrated in Figure 9. Among the fields within this *Command* message are the `read(10)` opcode `[0x28 0x0a]`, the requested data address offset, the transfer size, and a unique tag. The file contents are then provided in subsequent *Data* block(s) through one or multiple (depending on size) IN endpoint 1 transactions. The device indicates transfer completion *Status* through a subsequent IN endpoint 1 transaction. The *Status* message must include the same unique tag issued in the *Command* message.

At HS, the maximum transfer size for a data block is 512 bytes. When amounts of data in excess of this are requested it is all transmitted over multiple successive IN transactions. For brevity, any mentions of IN or OUT communications hereon refer to those over endpoints 1 and 2, respectively.

Further Modifications to the Injection Platform. Operating with awareness of the extra communication protocol layer of MSD victims requires some higher-level control functionality on the part of our platform. Directly in the platform hardware we configure ad hoc microcontroller application function to monitor downstream OUT messages to the victim and trigger an internal signal upon detection of the SCSI `read(10)` command. We also make it store the transmission size requested and the unique message tag.

We program it to construct injection packets consisting of the *g* character (`0x67`) (arbitrarily chosen), with the final packet being zero-padded to 512 bytes in length. It determines how many *g* characters to put in the injected packet(s) from the size field in the triggering `read(10)` command packet.

Some additional injections after the data is sent are required to fulfil exchanges that complete the file transfer. We make our platform inject a *Status* transmission (with the registered messaged tag) in response to the IN token following the last *Data* transmission. We program the platform to then acknowledge the subsequent OUT TUR messages (initial ACK response to the PING token then to OUT message) sent to the victim.

Experimental Setup. We configure this injection platform to present itself as a serial communications device operating at HS. It monitors HS OUT communications to other connected devices and once triggered, injects in response to the subsequent IN tokens addressed to the victim. The injection platform and victim flash drive attach to a common hub (previously found vulnerable) which connects to a Windows host through our protocol analyser. We prepare a text file in the victim file system consisting of several different characters.

Results. The attack is consistently successful since our injector consistently undercuts the MSD responses, and the host terminates the transfer leaving our injected packet contents on the host machine. We confirm the result with the analyser capture. Figure 10 shows bus communications during our file transfer hijack. After an ACK is injected in response to the TUR command, the driver concludes the file transfer. A key takeaway is that a malicious actor may need to characterise

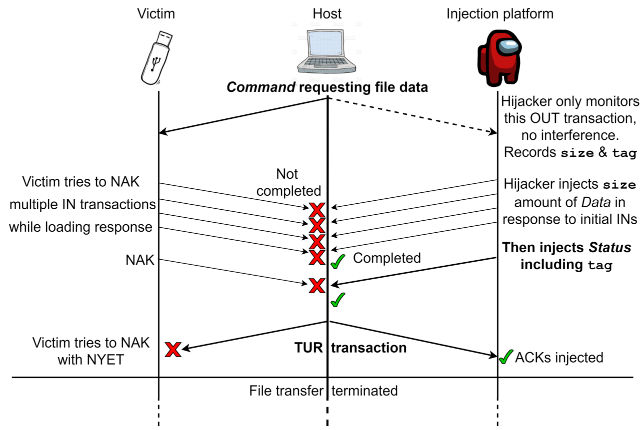

Figure 10: File transfer hijack communications. Each complete arrow represents a completed 3-stage USB transaction. The dotted line arrow represents a transaction that is merely observed by the injection platform.

the behaviour of target host system drivers to perform attacks against victims communicating with higher level protocols.

Operating System Image Compromise in USB Boot. We apply our file transfer hijacker to compromising a Kali Linux OS image transferred when a host boots from USB. Our attack changes the partition selection option labels that appear in the boot menu, switching the 'encrypted persistence' option with the 'persistence' option.

To do this, we performed a boot from the victim and recorded all traffic. In the capture we identify the `grub.cfg` file contents which happen to be transferred at the 17th IN transaction (and in plaintext) following a request for data at a certain address. We store the requested address in our platform and alter the injection trigger signal. The trigger signal is set after our platform has observed 16 IN tokens following a `read(10)` *Command* requesting the stored address. We inject a modified version of the genuine packet.

The compromise is successful. This case is different to the previous file transfer hijack since we are injecting in response to an IN token which is known to be met with a victim DATA packet instead of a NAK. Our injected data transmission arrives at the host earlier than the victim's and the host later ACKs the victim for the injected transmission, so the victim does not think it needs to resend the data.

Compromise in exactly this manner demands a strong attacker model since it requires knowledge of the victim data transfers that achieve a USB boot, and of course the content transferred. Nonetheless, it demonstrates that using our injection exploit we can compromise the boot image. In a more realistic attack scenario, an injection platform could be configured with the complete capability of a flash drive (while still presenting a benign device) and use injection to take over a victim connection entirely to force its own image onto the host. This is akin to how we previously continued to inject in subsequent communications to terminate a hijacked file transfer.

8 Circumventing Authorisation Policies

Due to the *trust-by-default* nature of USB, attack-capable devices only need access to typical, unprotected computer systems to feed keystroke commands or transfer malicious files. Attacks of this kind can usually be performed without injections. However as previously mentioned, injection is useful against protective measures that can govern the authorisations afforded to connected devices, since these policies are enacted at higher level communication layers than that where injection occurs. Authorisation policies trust and process messages that arrive from the host USB controller which, as we have shown, can be wrongly attributed. Such policies are widely adopted protections according to a survey carried out in 2019 [5] which found over 40% of organisations to use such policies.

Testing USB Authorisation Policies. We test our platforms against various authorisation policies to confirm that they can be circumvented by injection. Depending on how the policies can be set, for testing we either explicitly allow only a trusted victim to provide input while implicitly blocking all other devices, or we explicitly block our injection platform, allowing all else. In cases where it is possible to both explicitly allow and block certain devices we do so. With the policy in operation, we then attempt injection. The tested policy solutions and the policies are:

USBFILTER. USBFILTER [47, 49] is a packet-level access control system which can be linked to the Linux kernel. Through its use, packet filtering rules can be applied to the level of allowing or blocking certain device interfaces, also enabling restriction of interaction between device interfaces with certain applications/processes running on the host. In testing USBFILTER, we allow the trusted devices and block all interaction with our injection platforms.

GoodUSB. GoodUSB [46, 48] instruments the USB stack to let users moderate which drivers can be loaded for a device based on what functionality they expect, using a popup menu that appears when they plug it in. We only allow the victims normal use of their expected drivers in testing. Injection can exploit those victim interfaces both when the attack platforms have been allowed as their other benign device types and when they are not authorised for use.

USBGuard. USBGuard [55] is a device access control package. This software gives users options to ‘allow’, ‘block’, and ‘reject’ communication with certain devices. We allow our victims and perform testing where we block and where we reject the injection platforms.

Oracle VirtualBox. the Oracle VM VirtualBox extension pack [39] supports the use of USB devices in a virtual machine’s guest OS. The extension lets users maintain a list of devices to be allowed or blocked for use in the VM. We explicitly allow the trusted device and block the injection platform.

USB-Lock-RP. USB-Lock-RP [1] allows users to selectively allow only certain devices to work on a host. We allow only our trusted victim flash drive/keyboard to operate when testing. We tested both the free version and a licensed version that Advanced Systems International provided us.

Experimental Results. We successfully bypass all tested policies. We believe it is reasonable to claim that injection can be used to bypass all authorisation policies implemented anywhere in the USB software stack.

Device Fingerprinting Invariant. Device authorisation protections can be bypassed irrespective of the fingerprinting mechanisms they use to identify devices. Fingerprinting is most commonly based on *VendorID* (VID) and *ProductID* (PID) identifiers [52] supplied by devices during enumeration, however some mechanisms leverage other information like packet timings [13] or device electromagnetic emanations [20]. As we have demonstrated with selective injection, we can allow victim devices to operate as normal during such fingerprinting processes so that they are correctly identified.

9 Limitations, Future Work and Countermeasures.

Targeting Additional Devices. In its current form, our attack is limited to targeting communications of USB 2.0 and 1.x victim devices. This is due to our threat model’s dependence on monitoring downstream communications, which are only broadcast in these protocol versions, to stimulate injection responses. However, devices using these protocol versions continue to be highly relevant, as keyboards and other HIDs will continue to be manufactured to the 1.x standard.

The injection platforms created in this work have been made to demonstrate specific use cases attacking keyboard and mass storage device communications. Our file hijacker for example used specific knowledge of the mass storage device class protocol to achieve desired effects. We leave the task of implementing attacks against other device types and classes to future work.

Attacking USB 3.x Traffic. USB 3.x systems use point-to-point routing which prevents direct off-path communications monitoring, albeit inadvertently since this was introduced as a power saving measure to reduce unnecessary signal transmission in hubs and token address processing at devices. We do note however, that since USB 3.x hubs incorporate side-by-side SuperSpeed and 2.0 hubs for device backward compatibility [17], the 2.0 (or 2.1) hub within is attacked by our injection exploit, making these 3.x hubs susceptible.

It may yet be possible to inject USB 3.x traffic by transmitting in response to probes that an attacker indirectly monitors, perhaps from signal crosstalk leakages. Future work can try to target USB 3.x victims with a similar attack model and investigate mechanisms for off-path monitoring of downstream traffic in 3.x hubs.

Mitigations. A straightforward countermeasure is to use hubs that block transmission when a collision is detected. This is explicitly mentioned as a design option in the USB 2.0 specification however our results show the majority of USB 2.0 hubs do not implement this option. Conversely, the majority of USB 2.1 hubs (i.e. USB 2.0 hubs within 3.x hubs) do. Although collision detection prevents our attack, it does not address the core vulnerability as it relies on the trusted device causing a collision. Nonetheless, collision detection should be effective so long as the trusted device does not malfunction or have a malfunction induced by other means.

Further, adding capability to interpret the garbles sent upstream indicating collision detection in a host's software stack would mean the system user can be alerted that either an injection or malfunction is potentially occurring. Host controller chips must be designed to pass on collision information to the USB software stack for user alerts to work.

For vulnerable hubs, physical separation of devices from USB port lines through proxies and/or port blocking add-ons can be used to prevent injection and bus sniffing attacks.

10 Conclusions

In this paper we present an off-path transmission injection attack on the integrity of USB communications, the first of its kind. This provides us with a complete picture of the threat off-path devices can pose. We find the majority of USB 2.0 hubs to be vulnerable as they enable injection, and we discovered that a small proportion of USB 3.x hubs are vulnerable. By using our injection technique, an attacker can circumvent device authorisation policies enforced in a computer's software stack to exploit any communication channels that are trusted according to the user-configured policies. We circumvent all policies tested. We further demonstrate two attack scenarios: keystroke command injection and hijacking file transfers. The former allows us to inject malicious commands, and the latter allows us to compromise an OS image when booting from a trusted USB flash drive.

While our attacks do require a malicious device connected to the target system, we argue that these attacks do pose a non-trivial threat in some scenarios, especially for high-security USB applications. As current generation of USB hardware cannot be updated to prevent command injection, we leave the task of designing and deploying USB usage policies for preventing injection attacks to future work.

Acknowledgements

We thank the reviewers from the USENIX Security Program Committee for their insightful feedback, this paper was greatly improved across revisions based on their comments.

This work was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award number DE200101577; an ARC Discovery Project number DP210102670; the Blavatnik ICRC at Tel-Aviv University; the Defence Science and Technology Group (DSTG), Australia under Agreement ID10620; the National Science Foundation under grant CNS-1954712 and gifts from AMD, Intel, and Qualcomm.

References

[1] Advanced Systems International. USB-Lock-RP. https://www.usb-lock-rp.com/.

[2] Sebastian Angel, Riad S. Wahby, Max Howald, Joshua B. Leners, Michael Spilo, Zhen Sun, Andrew J. Blumberg, and Michael Walfish. Defending against malicious peripherals with Cinch. In *USENIX Security*, pages 397–414, 2016. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel.

[3] *Universal Serial Bus 3.2 Specification, Rev 1.0*. Apple Inc., Hewlett-Packard Inc., Intel Corporation, Microsoft Corporation, Renesas Corporation, STMicroelectronics, and Texas Instruments, September 2017.

[4] *Universal Serial Bus 4 (USB4™) Specification, Version 1.0*. Apple Inc., HP Inc., Intel Corporation, Microsoft Corporation, Renesas Corporation, STMicroelectronics, and Texas Instruments, August 2019.

[5] Apricorn. Apricorn report reveals employers' shortcomings in mitigating the risks of non-secure USB drives. https://apricorn.com/apricorn-report-reveals-employers-shortcomings-in-mitigating-the-risks-of-non-secure-usb-drives/, 2019.

[6] Architecture Technology Corporation. USB Sentry. https://www.atcorp.com/products/usbsentry/.

[7] John Clark, Sylvain Leblanc, and Scott Knight. Compromise through USB-based hardware Trojan horse device. *Future Gener. Comput. Syst.*, 27(5):555–563, 2011. doi: 10.1016/j.future.2010.04.008.

[8] COMXI Co. USB physical security. https://www.smartkeeperworld.com/#/usb-physicalsecurity.

[9] ComBlock. USB2SOFT USB 2.0 device SIE. https://www.comblock.com/zc/index.php?main_page=product_info&products_id=20.

[10] *Universal Serial Bus Specification, Rev 1.0*. Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, and Northern Telecom, January 1996.

[11] *Universal Serial Bus Specification, Rev 2.0*. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, April 2000.

[12] *Universal Serial Bus Specification, Rev 1.1*. Compaq, Intel, Microsoft, and NEC, September 1998.

[13] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. Time-print: Authenticating USB flash drives with novel timing fingerprints. In *IEEE SP*, pages 88–103, 2022. doi: SP46214.2022.9833595.

[14] Monta Elkins. Universal RF USB Keyboard Emulation Device URFUKED. https://defcon.org/images/defcon-18/dc-18-presentations/Elkins/DEFCON-18-Elkins-Universal-RF-Keyboard.pdf.

[15] Globotron Developments Ltd. Armadillo hardware firewall USB 2.0. https://globotron.nz/products/armadillo-hardware-usb-firewall.

[16] Travis Goodspeed. Python USB device emulation. http://goodfet.sourceforge.net/hardware/facedancer21/.

[17] *Universal Serial Bus 3.0 Specification, Rev 1.0*. Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, ST-NXP Wireless, and Texas Instruments, November 2008.

[18] *Universal Serial Bus 3.1 Specification, Rev 1.0*. Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, Renesas Corporation, ST-Ericsson, and Texas Instruments, July 2013.

[19] HighSecLabs. eLock USB Lockdown. http://www.highseclabs.com/products/?pid=92.

[20] Omar Adel Ibrahim, Savio Sciancalepore, Gabriele Oligeri, and Roberto Di Pietro. MAGNETO: Fingerprinting USB flash drives via unintentional magnetic emissions. *ACM Trans. Embed. Comput. Syst.*, 2020. doi: 10.1145/3422308.

[21] INT3.CC. The Original USB Condom. https://int3.cc/collections/usb/products/usbcondoms.

[22] Intel Corporation. *eXtensible Host Controller Interface for Universal Serial Bus (xHCI) Requirements Specification, Revision 1.2*, May 2019.

[23] Ivanti. Ivanti Device Control. https://www.ivanti.com.au/products/device-control.

[24] Jeffrey Robert Jacobs. Measuring the effectiveness of the USB flash drive as a vector for social engineering attacks on commercial and residential computer systems. Masters thesis, Embry-Riddle Aeronautical University, 2011.

[25] Robert E. Jenkins. FPGA-USB-V2 project. https://xess.com/projects/fpga-usb-v2-project/, August 2009.

[26] Samy Kamkar. USBdriveby. http://samy.pl/usbdriveby/.

[27] Keelog. KeyGrabber USB. https://www.keelog.com/usb-keylogger/.

[28] Kensington. USB port blocker. https://www.kensington.com/p/products/data-protection/usb-port-lock-security/usb-port-blocker-cable-guard-horizontal/.

[29] KeyCarbon. KeyCarbon USB. https://www.keycarbon.com/products/keycarbon_usb/overview/.

[30] David Kierznowski. BadUSB 2.0: USB man in the middle attacks. Masters thesis, Royal Holloway University of London, 2016.

[31] Darren Kitchen. USB rubber ducky. https://github.com/hak5darren/USB-Rubber-Ducky/wiki.

[32] Hao Liu, Riccardo Spolaor, Federico Turrin, Riccardo Bonafede, and Mauro Conti. USB powered devices: A survey of side-channel threats and countermeasures. *High-Confidence Computing*, 2021. doi: 10.1016/j.hcc.2021.100007.

[33] Mark Mamchenko and Alexey Sabanov. Exploring the taxonomy of USB-based attacks. In *Management of large-scale system development (MLSD)*, 2019. doi: 10.1109/MLSD.2019.8910969.

[34] Microchip. 32-bit PIC® and SAM Microcontrollers. https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus.

[35] Microsoft. Microsoft Windows Embedded 8.1 Industry. USB Filter (Industry 8.1). https://msdn.microsoft.com/en-us/library/dn449350(v=winembedded.82).aspx.

[36] Hessam Mohammadmoradi and Omprakash Gnawali. Making whitelisting-based defense work against BadUSB. In *ICSDE*, pages 127–134. ACM, 2018. doi: 10.1145/3289100.3289121.

[37] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. A transparent defense against USB eavesdropping attacks. In *EUROSEC*, pages 6:1–6:6, 2016. doi: 10.1145/2905760.2905765.

[38] Nir Nissim, Ran Yahalom, and Yuval Elovici. USB-based attacks. *Comput. Secur.*, 70:675–688, 2017. doi: 10.1016/j.cose.2017.08.002.

[39] Oracle. VM VirtualBox extension pack v6.1.22. https://www.virtualbox.org/wiki/Downloads.

[40] Dung Vu Pham, Ali Syed, and Malka N. Halgamuge. Universal serial bus based software attacks and protection solutions. *Digit. Investig.*, 7(3–4):172–184, 2011. doi: 10.1016/j.diin.2011.02.001.

[41] *UTMI+ Specification Rev 1.0*. Phillips, ARC, Cypress, Synopsys, Innovative, MCCI, ST Microelectronics, Mentor Graphics, TransDimension, and Portplayer Inc, February 2004.

[42] Seagate. *SCSI Commands Reference Manual*, October 2016.

[43] Dominic Spill. USB Proxy. https://github.com/usb-tools/USBProxy-legacy.

[44] Yang Su, Daniel Genkin, Damith Chinthana Ranasinghe, and Yuval Yarom. USB snooping made easy: Crosstalk leakage attacks on USB hubs. In *USENIX Security*, pages 1145–1161, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/su.

[45] *On-The-Go and Embedded Host Supplement to the USB Revision 3.0 Specification, Revision 1.1*. Texas Instruments, Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, Renesas Corporation, and ST-Ericsson, May 2012.

[46] Dave (Jing) Tian. GoodUSB. https://github.com/daveti/GoodUSB, February 2015.

[47] Dave (Jing) Tian. usbfilter. https://github.com/daveti/usbfilter, July 2015.

[48] Dave (Jing) Tian, Adam Bates, and Kevin Butler. Defending against malicious USB firmware with GoodUSB. In *ACSAC*, pages 261—270, 2015. doi: 10.1145/2818000.2818040.

[49] Dave (Jing) Tian, Nolen Scaife, Adam Bates, Kevin R. B. Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *USENIX Security*, pages 415–430, 2016. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tian.

[50] Dave (Jing) Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin R. B. Butler. SoK: "plug & pray" today - understanding USB insecurity in versions 1 through C. In *IEEE SP*, pages 1032–1047, 2018. doi: 10.1109/SP.2018.00037.

[51] Matthew Tischer, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein, and Michael Bailey. Users really do plug in USB drives they find. In *IEEE SP*, pages 306–319, 2016. doi: 10.1109/SP.2016.26.

[52] USB ID Repository. http://www.linux-usb.org/usb-ids.html.

[53] USB Implementers Forum. Device class definition for human interface devices (HID). https://www.usb.org/sites/default/files/hid1_11.pdf, May 2001.

[54] USB Implementers Forum. USB-IF statement regarding USB security. https://www.usb.org/press/USB-IF_Statement_on_USB_Security_FINAL.pdf, August 2014.

[55] USBGuard. https://usbguard.github.io/.

[56] USBKill. https://usbkill.com/.

Appendix

In this appendix we provide further tables and figures that augment and complete the information in the paper.

Victim Devices Used in Testing. Table 2 lists the devices we used as the victim devices for injection tests. Overall, we used six victim devices. For the USB 1.x experiments we used two keyboards, one operating at LS and the other at FS. For the USB 2.0 experiments we used four different USB disks.

Hubs Tested. Table 3 lists all hubs we tested. The model describes the packaging of the hub. **Type** indicates whether the hub is a standalone device (S) or embedded within a host system (E), for example embedded within a motherboard. The ticks in the columns labelled by speed mode (**1.x** – LS/FS and **2.0** – HS) indicate hubs that were found vulnerable to injection in that speed mode. *VendorID* (***VID***), *ProductID* (***PID***), and *bcdDevice* (***bcdDev*** – device version) are descriptors provided by the hub during enumeration with a host. **TT** indicates whether the hub is a multi-TT (M) or single-TT (S) system. Also note that for the 3.0 hubs, we refer to the internal 2.0 hub chip descriptors.

In some cases duplicate chips are listed, for instance the 2.0 hub chip within the vulnerable Alogic 3.0 hub has the same IDs as a chip from a non-vulnerable model - the Sabrent 3.0 hub. Despite the identical IDs we found these two chips to present slightly different descriptor sets which indicates they may have undergone different configuration when ported into

Table 2: USB victim devices used to test injection

Device	*VendorID*	*ProductID*	*bcdDevice*	Speed
Dell Quietkey Keyboard	0x413C	0x2106	0x0101	LS
Corsair K55 RGB PRO Gaming Keyboard	0x1B1C	0x1BA4	0x0101	FS
Emtec USB DISK 2.0	0x6557	0x0121	0x0100	HS
Silicon Motion, Inc. USB Flash Disk	0x090C	0x1000	0x1100	HS
SanDisk U3 Cruzer Micro Flash Drive	0x0781	0x5406	0x0200	HS
SanDisk Cruzer Blade Flash Drive	0x0781	0x5567	0x0100	HS

Table 3: USB hub models tested. **Version** indicates USB version. **Type** indicates standalone device (S) or embedded (E) within a host system, e.g. motherboard. *VendorID* (***VID***), *ProductID* (***PID***), and ***bcdDevice*** (device version) are hub-provided descriptors. **TT** indicates single- (S) or multi-TT (M). Ticks under **1.x** or **2.0** indicates vulnerability to injection in those modes.

USB Hub Model	Version	Type	2.0 Chip Vendor (*VID*)	*PID*	*bcdDev*	TT	1.x	2.0
1PortUSB Network with 3Port	2.0	S	Terminus Technology Inc. (0x1A40)	0x0101	0x0100	S		
D-Link DUB-H7 7-Port	2.0	S	Terminus Technology Inc. (0x1A40)	0x0101	0x0111	M		✓
D-Link DUB-H7 7-Port	2.0	S	D-Link Corp. (0x2001)	0xF103	0X0100	S	✓	✓
Dell EMC	2.0	S	AMECO Technologies (0x214B)	0x7000	0x0100	S		
Gigabyte B550 AORUS ELITE V2 Motherboard	2.0	E	Genesys Logic Inc. (0x05E3)	0x0608	0x8536	S	✓	
Gigabyte H470 HD3 Motherboard	2.0	E	Genesys Logic Inc. (0x05E3)	0x0608	0x8536	S	✓	
J. Burrows High-Speed	2.0	S	Terminus Technology Inc. (0x1A40)	0x0101	0x0111	M		✓
Micro-Star PRO Z690-A Motherboard	2.0	E	Genesys Logic Inc. (0x05E3)	0x0608	0x6070	S	(✓)	
Speedlink Barras Supreme Hub and Sound Card	2.0	S	Genesys Logic Inc. (0x05E3)	0x0608	0x8536	S	✓	
Startech Industrial 4 Port Mountable	2.0	S	NEC Corp. (0x0409)	0x005A	0x0100	S	✓	
Targus	2.0	S	Genesys Logic Inc. (0x05E3)	0x0608	0x8537	S	✓	
Tripp Lite	2.0	S	Terminus Technology Inc. (0x1A40)	0x0101	0x0111	M		✓
Unlabelled operating at LS/FS	2.0	S	Genesys Logic Inc. (0x05E3)	0x0606	0x0702	S	✓	N/A
Unlabelled 'Slim' hub operating at LS/FS	2.0	S	Genesys Logic Inc. (0x05E3)	0x0606	0x0702	S	✓	N/A
Unlabelled	2.0	S	MOAI Electronics Corp. (0x14CD)	0x8601	0x0000	S		
Unlabelled	2.0	S	Terminus Technology Inc. (0x1A40)	0x0101	0x0111	M		✓
Alogic USB-C Fusion SWIFT 4-in-1	3.0	S	Genesys Logic Inc. (0x05E3)	0x0610	0x0655	S	✓	
Asmedia ASM107x	3.0	S	QNAP System Inc. (0x1C04)	0x2074	0x0100	M		
Belkin F4U090	3.0	S	Belkin International, Inc. (0x050D)	0x090B	0x5102	M		
Bonelk 4 Port	3.0	S	Realtek Semiconductor Corp. (0x0BDA)	0x5411	0x0101	M		
Channel+	3.0	S	VIA Labs Inc. (0x2109)	0x2813	0x0221	S		
HP	3.0	S	HP Inc. (0x03F0)	0x444A	0x0125	M		
Plugable	3.0	S	VIA Labs Inc. (0x2109)	0x2813	0x9011	S		
Sabrent	3.0	S	Genesys Logic Inc. (0x05E3)	0x0610	0x0655	S		
Satechi	3.0	S	VIA Labs Inc. (0x2109)	0x2817	0x0383	M		
Smart Sync & Charge	3.0	S	Genesys Logic Inc. (0x05E3)	0x0610	0x9226	M		
Targus	3.0	S	VIA Labs Inc. (0x2109)	0x2813	0x9011	S		
Ugreen	3.0	S	VIA Labs Inc. (0x2109)	0x2817	0x9013	M		
Wavlink	3.0	S	VIA Labs Inc. (0x2109)	0x2815	0x0704	M		

Table 4: USB product vendors and manufacturers contacted for disclosure

Company	Type	Report Sent	Most recent response to Report
Alogic	Hub vendor	✓	Technical team assessing report findings
D-Link	Hub vendor and manufacturer	✓	Asserted their products are specification compliant, technical team assessing report findings
Genesys Logic Inc.	Hub manufacturer		No response to initial contact
Gigabyte Technology	Hub vendor	✓	Technical team assessing report findings
J. Burrows (Officeworks)	Hub vendor		No response to initial contact
Oracle (VM Virtual Box)	Software vendor		No response to initial contact
NEC Corporation	Hub manufacturer		No response to initial contact
Speedlink	Hub vendor	✓	Not concerned because the product has been discontinued
Startech	Hub vendor	✓	Technical team assessing report findings
Targus	Hub vendor	✓	Escalated with senior management
Terminus Technology Inc.	Hub manufacturer	✓	Not concerned, they say it is irrelevant with their hubs
Tripp Lite	Hub vendor	✓	Not concerned because their product is not a network device
USB-Lock-RP	Software vendor	✓	Acknowledged findings

their USB 3.0 hub. For this reason we have retained all cases of duplicate chips among different hub products.

Responsible Disclosure. Table 4 lists the vendors we notified and summarises their responses. To ensure that disclosure is coordinated, the initial contact did not include a bug report. Instead, it sought an agreement not to disclose until a mutually agreed disclosure date. Of the 13 vendors we approached, four failed to respond to the initial contact, despite repeated attempts. The other nine agreed to the terms and received the report. Only four of the vendors responded to the contents of the report, three of which claim not to be concerned, whereas one vendor acknowledged the finding.

Injection Platform Hardware. Last, we include images of the target hardware to which we ported our injection platform implementations. Figure 11 shows the 1.x injector hardware described in Section 5.1. This includes additional external circuitry required to interface directly to the USB lines. Figure 12 shows the 2.0 injector hardware described in Section 5.2. The hardware comprises a target FPGA configured with the USB device core, and an external dedicated PHY module connected by wires over UTMI+.

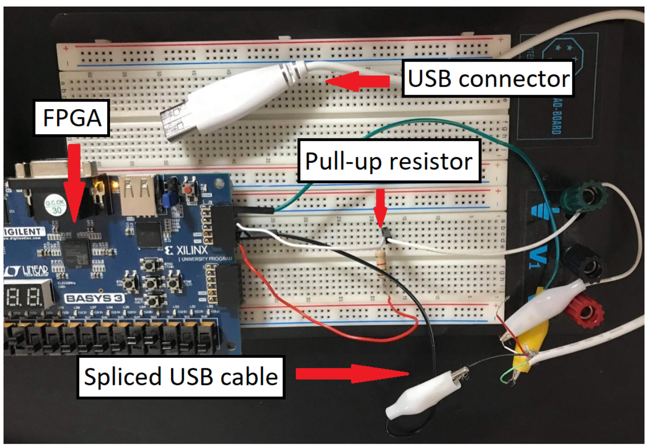

Figure 11: USB 1.x target hardware

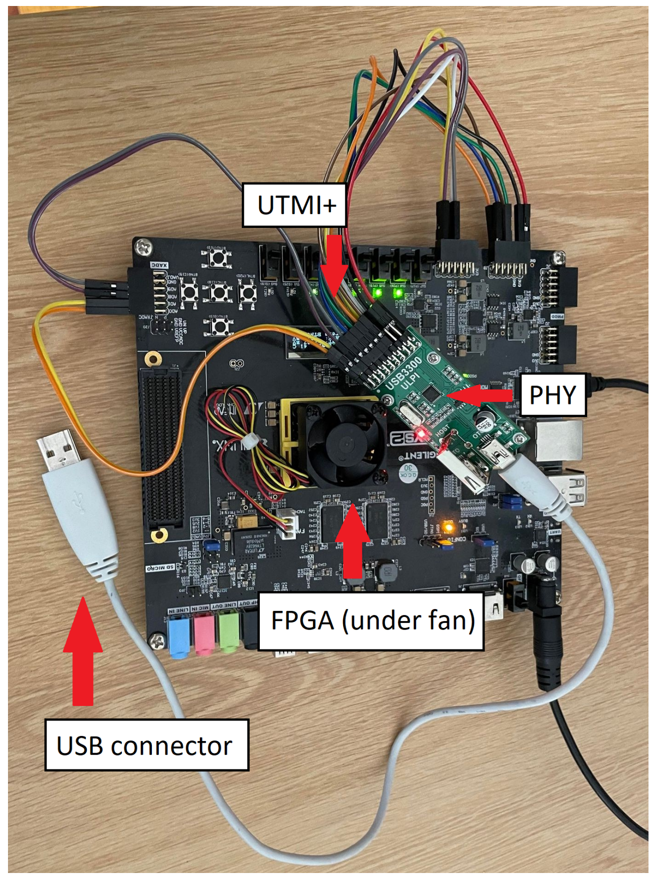

Figure 12: USB 2.0 target hardware