2009 Third International Conference on Network and System Security

Hardware Trojan Horse Device based on Unintended USB Channels

John Clark, Sylvain Leblanc and Scott Knight
Electrical and Computer Engineering Department
Royal Military College of Canada
Kingston, Ontario, Canada
{*john.clark, sylvain.leblanc, knight-s*}*@rmc.ca*

Abstract

This paper discusses research activities that investigated the risk associated with USB devices. The research focused on identifying, characterizing and modelling unintended USB channels in contemporary computer systems. Such unintended channels can be used by a USB Hardware Trojan Horse device to create two way communications with a targeted network endpoint, thus violating the integrity and confidentiality of the data residing on the endpoint. The work was validated through the design and implementation of a proof of concept Hardware Trojan Horse device that uses two such unintended USB channels to successfully interact with a target network endpoint to compromise and exfiltrate data from it.

1. Introduction

The USB protocol is ubiquitously on modern computer systems such as servers and network endpoints[1]. USB peripherals offer network endpoints a wide variety of functionality, from storage to Human Interface Devices. The USB Specification defines a single physical interface and base protocol to be used for all USB devices [1]. USB devices are Plug and Play, meaning that a contemporary computer system contains the driver software necessary to configure a newly attached USB device without user or system administrator intervention.

The very properties that make USB attractive to users (dynamic attachment, automatic configuration and the use of a single bus for a wide range of devices) can also turn USB into a viable attack vector into a network endpoint. Even when used as intended, USB can pose significant risks to confidentiality, as is the case when using USB storage devices to exfiltrate large amounts of data from a network endpoint. However, USB can also affect the integrity of information on the network endpoint. Attackers have exploited Plug and Play, USB Flash Drives and other usability features (such as AutoRun and AutoPlay) to inject Software Trojan Horses in network endpoints [2], [3].

Endpoint Security Solutions have been introduced to help protect contemporary computer systems from the risks of such intended USB communication. Endpoint Security Solutions work by extending an access control list to certain classes of devices such as mass storage devices, imaging devices, PDAs, printers and communication ports. The Endpoint Security Solutions examined in this research did not generally regulate Human Interface Devices such as mice, keyboards and speakers [4], [5], [6], [7]. Our work demonstrates that an attacker can make use of certain USB devices, not controlled by Endpoint Security Solutions, to create a Hardware Trojan Horse device. The Hardware Trojan Horse device can be attached to a target network endpoint as a replacement for the existing keyboard. Providing keyboard functionality allows the Hardware Trojan Horse to keylog user credentials as well as to upload and cause the execution of arbitrary code. Such arbitrary code leaves the network endpoint vulnerable to a wide range of known malicious attacks. More importantly, the ability to execute arbitrary code allows the Hardware Trojan Horse device to create unintended USB channels in order to establish two-way communications between itself and the network endpoint.

We hypothesize that the USB protocol carries the risk of unintended USB channels. We further note that this risk has not been well researched to date.

An unintended USB channel is one where the USB protocol is used to communicate in a way not anticipated by the USB protocol. As an example consider the communications between a USB keyboard and a network endpoint. USB keyboards normally use two *intended USB channels* to communicate with a network endpoint[2]. One channel exists to transfer data in the form of key presses from the keyboard to the network endpoint. The other channel exists to communicate the state of certain modifiers (e.g. `Caps Lock`, `Number Lock` and `Scroll Lock`) from the network endpoint to attached USB keyboards. An application on the network endpoint could create an *unintended USB channel* by causing the transmission of LED Status messages

1. We use the term network endpoint to refer to production workstations attached to the corporate LAN. The USB specifications describe a similar term *device endpoint* which refers to the source or sink of data on a USB device.

2. The USB Specification [1] and Axelson [8] provide a good description of intended USB channels.

978-0-7695-3838-9/09 $26.00 © 2009 Crown Copyright
DOI 10.1109/NSS.2009.48

IEEE computer society

in a sequence that has meaning to the attached keyboard other than the toggling of the keyboard LEDs.

1.1. Related Work

The open literature does not discuss unintended USB channels, but the following bear some relevance on our research.

Because the USB protocol relies on devices to properly identify themselves during enumeration, a *USB Meta-Device* could be programmed to identify itself as any USB device [9]. In this way, the USB Meta-Device could be configured to represent itself as a device associated with a vulnerable driver loaded on the network endpoint. Our approach differs from the USB Meta-Device in that it does not require the presence of a vulnerable driver on the network endpoint.

There are risks associated with the consented use of USB [10] where legitimate users allow a foreign USB device to connect to the network endpoint, either voluntarily or because they have been duped [2]. In fact, Endpoint Security Solutions have been developed to mitigate this risk. Our approach differs in that it can still be used in the presence of Endpoint Security Solutions, as those generally do not regulate Human Interface Devices.

The idea of a *Keyboard Jitterbug* was introduced in 2006 [11]. Such a device can be inserted between a keyboard and a network endpoint and act as a keylogger. This work is interesting in that it demonstrates that a device that provides keyboard functionality can also be used to exfiltrate data from a network endpoint. However, it is different from our approach because it requires that the network endpoint maintain an interactive session through the Internet. Our Hardware Trojan Horse device is able to exfiltrate information from a network endpoint without having to rely on the network.

Finally, while there is no known research into unintended USB channels, there is extant *covert channel* research. A covert channel can be defined as "an enforced, illicit signalling channel that allows a user to surreptitiously contravene the security policy and unobservability requirements of the system" [12]. As our unintended USB channels can be used to exfiltrate data from the network endpoint, they are similar to covert channels, even if we do not stress their unobservability. We use throughput to characterize the seriousness of the risk associated with unintended USB channels, because it is a recognized characteristic of such channels.

The remainder of this document is separated in three main sections. Section 2 will discuss two unintended USB channels, models of their throughput and the experiments we conducted to verify those models. Section 3 will discuss the validation of our research aim, carried out by developing and implementing a proof of concept Hardware Trojan Horse device. Finally Section 4 will conclude with a discussion of other relevant factors that should be considered when analyzing the risks of unintended USB channels and potential avenues for future work.

2. Two Unintended USB Channels

As we will discuss later, our work is based on the premise that a Hardware Trojan Horse device must be capable of uploading applications to create the unintended USB channel. Any USB device connected to a network endpoint presents an interface defining certain connection parameters during enumeration. In order to use unintended USB channels to exfiltrate data from a network endpoint, a Hardware Trojan Horse device would have to be successfully enumerated by a network endpoint. This requires that the device interface presented to the network endpoint be recognized as legitimate by an Endpoint Security Solution.

In approaching this research, we first identified a number of potential unintended USB channels based on devices that are not well regulated by Endpoint Security Solutions. These potential channels were classified into two categories: *User-Space* and *Kernel-Space* channels. In User-Space channels, the USB protocol is used as implemented to create unintended USB channels; in Kernel-Space channels, the Kernel is subverted in oder to modify the USB System Software on a network endpoint to create unintended USB Channels. Our research focused exclusively on User-Space unintended USB channels.

Three types of transfers used by potential User-Space channels were examined: Control, Interrupt and Isochronous Transfers. These were examined to determine if they could form the basis of an unintended USB channel. Bulk transfers, the fourth type of USB transfer, were discounted as they are associated with devices that would be well regulated by Endpoint Security Solutions.

Control and Isochronous transfers were further examined because devices associated with them are not currently viewed as a risk for the exfiltration of data, and are not considered in current risk mitigation strategies. As discussed, a Hardware Trojan Horse must be able to enumerate in the face of an Endpoint Security Solution. Keyboards are considered essential peripheral device for a contemporary computer system, and are generally not regulated by Endpoint Security Solutions. We therefore identified a potential unintended USB channel based on a USB keyboard using Control Transfers. However, as USB keyboards are Low-Speed devices, an unintended USB channel based on a keyboard would likely have limited throughput. To be effective, a Hardware Trojan Horse device would need a channel with higher throughput. This lead us to identify a second potential unintended USB channel based on Isochronous Transfers to a Full-Speed speaker. While speakers are not considered essential peripheral devices, they are generally

only considered able to receive audio data, and they are generally not regulated by Endpoint Security Solutions.

The examination of these two candidate channels entailed initial experimentation to better understand the data transfers. Following this, a mathematical model of the unintended USB channel's throughput was developed. The Throughput Model for each unintended USB channel was then verified through experimentation.

In order to perform the verification, a USB keyboard interface and a USB speaker interface were developed using PLX's Net2280 Resource Development Kit [13]. The Net2280 is a PCI card that can be made to appear as one of a number of USB devices. The keyboard interface was designed to provide keyboard functionality and keylog a user's credentials. Both interfaces were designed to decode data transmitted on the applicable unintended USB channel and record the elapsed time between transfers to compute throughput. The target network endpoint used for the verification and validation was a contemporary computer system running the Windows XP Professional Service Pack 2 operating system.

2.1. Keyboard LED Channel

USB keyboards are Low-Speed Human Interface Devices that use two intended USB channels to communicate with a network endpoint: one based on Interrupt Transfers and one based on Control Transfers. The Interrupt Transfer endpoint is unidirectional, with data flowing into the network endpoint, whereas the Control Transfer endpoint is bidirectional, making it a good candidate for the exfiltration of data from a network endpoint to a device identifying as a USB keyboard.

The USB Specification [1] and the USB Device Class Definition for Human Interface Devices [14] describe the format of data packets used for Control Transfers between a network endpoint and a USB keyboard. In particular, a Keyboard Output Report is described, which allows the manipulation of 3 bits mapped to the state of `Caps Lock`, `Scroll Lock` or `Num Lock` modifiers.

A Keyboard Output Report is generated by a network endpoint every time a keyboard transmits that a modifier key such as the `Caps Lock`, `Scroll Lock` or `Num Lock` key has been pressed. The network endpoint toggles the appropriate bit(s) in the Keyboard Output Report to represent the change, and transmits the report to all attached keyboards. Upon receipt of the report, attached USB keyboards toggle the physical LEDs corresponding to the changed modifier keys. Normally a Keyboard Output Report is only generated when there is a change in state of the modifier keys.

2.1.1. Initial Experimentation. The USB Specification indicates that a device has 5 seconds to handle a Control Transfer with a Data Stage [1]. This upper bound response time is not sufficient to allow us to precisely compute the throughput for the Keyboard LED Channel. Therefore, a series of experiments were designed and conducted to determine a representative response time for a device to handle Control Transfers with a Data Stage.

We created a VBScript file containing key press information to generate Keyboard Output Reports while imposing a delay between report generation. The imposed delay varied between 25 msec and 150 msec for each trial. The VBScript was executed on a target network endpoint and Keyboard Output Reports were observed using HHD's USB Monitor Professional [15]. The results of these initial experiments allowed us to determine that a delay of 109.5 msec between the generation of Keyboard Output Reports allowed for successful reception by the USB keyboard interface. When the delay was less than 109.5 msec, some of the Keyboard Output Reports were not received by the USB interface, a situation we called *deletion error*. The frequency of deletion errors increased as the delay between Keyboard Output Reports was shortened.

We also observed another condition where Keyboard Output Reports that were not generated by the network endpoint were observed at the USB keyboard interface; a situation we called *insertion error*. Insertion errors were infrequent during initial experimentation. Three inserted symbols were observed for 77 360 symbols transmitted, yielding a symbol insertion rate of 0.0039%. The elapsed time between the transmission of a legitimate symbol and an inserted symbol was observed to be significantly shorter than the elapsed time between successive legitimate symbols of 109.5 msec. Of the three insertion errors, two had elapsed times of 16 msec and one had an elapsed time of 31 msec.

This characteristic of insertion errors allowed us to locate them easily; it also allowed us to manually reconstruct the intended symbol by removing the inserted symbol from the communication stream. The inserted symbols occurred so infrequently that their effect on the Keyboard LED Throughput Model was ignored.

Being able to transmit three bits of information every 109.5 msec, yields a *Theoretical Throughput* for the Keyboard LED Channel of 3.42 bytes/sec.

2.1.2. Keyboard LED Channel Throughput Model. We selected a coding scheme in order to model the throughput of the Keyboard LED Channel. We chose a simple state based coding scheme, as shown in Figure 1. Our application generates Keyboard Output Reports representing fictitious changes to the state of the modifier keys to exfiltrate data, and not in response to user action as is normally the case. Because three bits of the report can be manipulated, our code allows for seven possible symbols to exfiltrate a 7 bit ASCII characters (the eight possible symbol corresponding to no change in state of the modifier keys is never generated). From Figure 1 we see that the state machine starts at

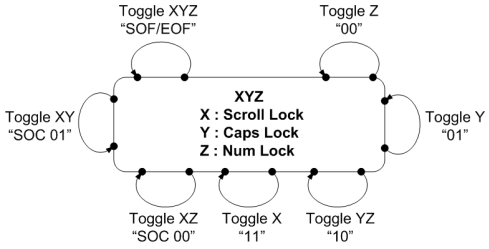

Figure 1. Keyboard LED Channel Coding Scheme

an arbitrary state for the modifier keys, XYZ: where X represents the current state of the `Scroll Lock` modifier, Y the current state of the `Caps Lock` modifier and Z the current state of the `Num Lock` modifier. A symbol can be generated by creating a Keyboard Output Report where one or more of the three available bits are toggled. Of the seven possible symbols, four carry two bits of information, two symbols carry one bit of information and are padded with a most significant bit of 0. These padded symbols are used to mark the Start of a Character (SOC). The seventh symbol, corresponding to toggling all three modifiers at once was used as a framing character (Start/End of Frame - SOF/EOF). For the purposes of this work, the start and end of a message was indicated by toggling the SOF/EOF symbol twice, resulting in an overhead of 1 byte for a message of any arbitrary size. In order to exfiltrate an ASCII message using the Keyboard LED Channel, each ASCII character requires four symbols. For example, in order to exfiltrate the ASCII character "S" (0101 0011 in binary), four symbols would be required: a "SOC 01", a "01", a "00" and a "11" symbol. Assuming a start state where only the `Num Lock` modifier is on (XYZ = 001), Keyboard Output Reports would be generated to toggle the `Scroll Lock` and `Caps Lock` (symbol 1), the `Caps Lock` (symbol 2), the `Num Lock` (symbol 3) and the `Scroll Lock` (symbol 4).

For a message of M bytes, the Keyboard LED Channel Throughput Model is then:

$$Keyboard\ LED\ Channel\ Throughput\ [bytes/sec] = \frac{M}{4 * (M + 1) * 0.1095} \quad (1)$$

Because of the imposition of overhead by the framing characters in our chosen coding scheme, the Keyboard LED Channel can never match the *Theoretical Throughput*. The Throughput Model is a limiting function of the message size and reaches a maximum achievable throughput of 2.283 bytes/sec.

2.1.3. Keyboard LED Channel Verification. An application, known as the Keyboard LED Coding Application, was designed and implemented in C++ that parsed a text file and generated a VBScript containing associated symbols (key press events to toggle modifier keys). The generated VBScript imposed a delay between successive symbols of 109.5 msec. The Keyboard LED coding application was placed on a target network endpoint, along with a 161 byte text file.

For this file size, the Keyboard LED Channel Throughput Model predicted a throughput of 2.269 bytes/sec. 20 exfiltrations were performed resulting in an average observed throughput of 2.274 bytes/sec. The observed throughput deviated from the modelled throughout by 0.22%. No inserted symbols were observed during the verification of the Keyboard LED Channel throughput model. Based on these results, we considered the Keyboard LED Channel Throughput Model to be verified as accurate. Note that the same experiment was used to validate our research aim in Section 3.3.

2.2. Audio Channel

USB speakers are Full-Speed Audio devices that have two device endpoints: a Control Transfer endpoint (similar to that described for a USB keyboard), and an Isochronous endpoint to receive a stream of data from a network endpoint. The Isochronous Endpoint is unidirectional, with data flowing out from the network endpoint.

The USB Specification describes the maximum Isochronous Transfer data packet size for a Full-Speed devices as 1023 bytes. It also specifies that a Full-Speed device, using Isochronous Transfers to communicate, can receive a data packet every frame, or every 1 msec. An unintended USB channel based on the Isochronous Transfers of 1023 bytes every 1 msec yields a *Theoretical Throughput* of 1023 kilobytes/sec.

The USB Device Class Definition for Audio Devices 2.0 [16] generally describes how a network endpoint marshals data to be sent to an audio device into structured blocks. The size of the block, BSize, is a product of the audio file's coded sample resolution, in bytes/sample, and of the number of channels.

$$BSize = Number\ of\ Channels\ * Sample\ Resolution \quad (2)$$

The `WAVEFORMATEXTESNIBLE` [17] audio format was chosen for this work, allowing for sample resolutions of 3 bytes/sample and four channels. This provided a `BlockSize` of 12 bytes. The number of blocks (`NumBlocks`) that are inserted into one Isochronous Transfer data packet also depends on the Sample Rate of the audio file. The `WAVEFORMATEXTESNIBLE` audio format allowed for a Sample Rate of 85 000 samples/sec for each channel, or 85 blocks of `BSize` per data packet transmitted every 1 msec.

Thus, using the characteristics for an audio file described

above, we obtained a data packet size of:

$$\begin{aligned} Audio\ Data\ Packet &= 85 * 12 \\ &= 1020\ bytes/packet \end{aligned} \quad (3)$$

2.2.1. Initial Experimentation. A `C++` application, known as the Audio Coding Application, was designed and implemented to create an appropriately formatted `WAVEFORMATEXTESNIBLE` audio file from a text file. The entire text file is directly transposed in the audio file's `Data SubChunk`.

For this work, with the Sample Rate set to 85 000 blocks per second (or 85 blocks per Isochronous Transfer data packet), and a `BSize` of 12 bytes/block, we can consider messages exfiltrated via the Audio Channel in terms of blocks. The number of blocks (`NumBlocks`) required for a message of M bytes is:

$$NumBlocks\ [blocks] = \frac{M}{BlockSize} \quad (4)$$

Based on our chosen actual data packet transmitted size and the Full-Speed speaker's periodicity, the maximum achievable throughput is 1020 kilobytes/sec.

A series of initial experiments were designed and conducted to examine the representation of the data that was transmitted over the Audio Channel. The generated audio file was played to an attached speaker device using the `Playsound()` API, and the data transmitted was observed using HHD's USB Monitor Professional. Two observations were made following these experiments: there was more data transmitted than was expected and there was some variability in the data transmitted.

The additional data observed was digital silence (0x00 or 0xFF) appended to the end of the data stream representing the encoded audio file. The size of the additional data was between 30 and 40 full packets. The variability in the amount of additional data, termed *End Stuffing*, was determined to be due to the network endpoint's operating system's buffering policy for audio data used to prevent device starvation [18]. An expression was developed to express End-Stuffing in terms of packets:

$$\begin{aligned} &End\ Stuffing\ [packets] = \\ &\quad 30 + 10 * \left(\left\lceil \frac{NumBlocks}{850} \right\rceil - \frac{NumBlocks}{850} \right) \end{aligned} \quad (5)$$

The data encoded in the audio file was occasionally modified in transmission, where characters received were - 0x01 of the encoded hexadecimal value in the audio file's `Data SubChunk`. We observed that this variability occurred infrequently, but we made no attempts to analyze its source, or its frequency of occurrence.

Because of the end stuffing of digital silence and the variability in the data transmitted, we chose a coding scheme that saw upper case ASCII letters, numbers and punctuation (69 possible characters) mapped to three symbols each in the range of 0x00 to 0xFF. Data variability and digital silence made it impossible to represent lower case letters using 256 symbols.

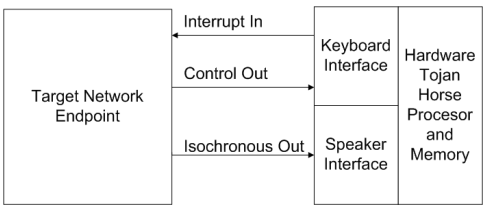

Figure 2. Hardware Trojan Horse Device

2.2.2. Mathematical Model of Theoretical Throughput. Taking End Stuffing in consideration, we developed the following expression for the transmit time (`TxTime`) of a message using the Audio Channel:

$$\begin{aligned} &TxTime\ [sec] = \\ &\quad 0.001 * \left(\frac{NumBlocks}{BlocksPerPacket} + End\ Stuffing \right) \end{aligned} \quad (6)$$

The Audio Channel Throughput Model is then given by the message size M divided by the time required to transmit it, including end-stuffing:

$$\begin{aligned} &Audio\ Channel \\ &Throughput\ [bytes/sec] = \frac{M}{TxTime} \end{aligned} \quad (7)$$

2.2.3. Audio Channel Verification. The audio coding application was placed on a target network endpoint along with a 40 800 byte text file. For a file of this size, requiring 100 full Isochronous data packets, the Audio Channel Throughout Model predicted a throughput of 784 615.3 bytes/sec. 20 exfiltrations were performed resulting in an average observed throughput of 784 695.2 bytes/sec. The observed throughput deviated from the model throughput by 0.01%. Based on these results, we considered the Audio Channel throughput model verified as accurate.

3. Validation

We designed and implemented a proof of concept USB Hardware Trojan Horse device, based on the Net2280 Resource Development Kit, to validate our hypothesis that unintended USB channels represent a risk to contemporary computer systems. A Hardware Trojan Horse device is similar to a software Trojan Horse, in that it can provide malicious functionality in addition to its apparent legitimate and intended use. However, Hardware Trojan Horse devices are different because they have their own processor and do not have to rely on the targeted network endpoint for processing.

A representation of the USB endpoints and interfaces of the proof of concept Hardware Trojan Horse device we

envisioned is given at Figure 2. The Proof of Concept Hardware Trojan Horse device presents two USB interfaces: a keyboard and a speaker, and it implements the unintended Keyboard LED channel (using the `Control Out` endpoint) and the unintended Audio Channel (using the `Isochronous Out` endpoint). We envisioned a Hardware Trojan Horse employed in the following scenario.

3.1. Representative Scenario

A network endpoint's keyboard is replaced by a Hardware Trojan Horse. The Hardware Trojan Horse functions as a keyboard, but it also logs the authorized user's authentication credentials. The Hardware Trojan Horse activates outside the organization's business hours and logs in to the network endpoint. It then uploads the applications necessary to create the Keyboard LED and Audio Channels between the network endpoint and the Hardware Trojan Horse, and applications that can open a tunnel and a back door to the Internet. The Hardware Trojan Horse then carries out its attack. It performs a search of the network endpoint's data, looking for files containing uploaded keywords and directing the results of the search to a text file. These search results, consisting of the path to files containing keywords, is exfiltrated to the Hardware Trojan Horse using the Keyboard LED Channel. The search result is analyzed by the Hardware Trojan Horse, which then causes the exfiltration of the files of interest from the network endpoint to the Hardware Trojan Horse using the Audio Channel. The exfiltrated files are further analyzed by the Hardware Trojan Horse, and one is chosen to be sent through the open Internet tunnel. Finally, the Hardware Trojan Horse opens a back door to the network endpoint and suspends the attack by returning the network endpoint to the state in which the authorized user had left it.

Two experiments were conducted in order to validate our research hypothesis. The first experiment, termed the *Upload Experiment*, observed the time necessary for the Hardware Trojan Horse to upload required applications. The second *Attack Experiment* observed the correct functioning of the Hardware Trojan Horse as it used both unintended USB channels to exfiltrate data from a network endpoint, processed the data and exercised its tunnelling and back door capabilities.

3.2. Upload Experiment

Four files were uploaded to a target network network endpoint, using the Hardware Trojan Horse's keyboard functionality: 1) the Keyboard LED Channel Coding Application which encodes the contents of a text file into a `VBScript` that generates Keyboard Output Reports; 2) the Audio Coding Application which encodes the contents of a text file into the `Data SubChunk` of a `WAVEFORMATEXTENSIBLE` audio file; 3) the `PlaySound()` application that plays an audio file to attached USB speakers; and 4) the tunnel tool that allows a target file to be tunnelled out from the network endpoint to the Internet, and allows a back door into the network endpoint. The size of the four files was over 193.8 KB.

The first three files were required to create the unintended USB channels from the target network endpoint to the Hardware Trojan Horse. The fourth file, a tunnel tool, was also included to demonstrate the ability of the Hardware Trojan Horse to cause the execution of more malicious code on the target network endpoint.

An available tool, EXE2VBS [19], was used to convert the four files from executables into a text representation, suitable for the keyboard interface to transmit one character at a time into a newly opened text document. The text representations contained the executable in hexadecimal representation wrapped in a `VBScripts` structure. When the VBScript was executed, the executable was unpacked and available for use.

The *Upload Experiment* measured the time required to upload and compile the files. It should be noted that the time to upload the files had much more impact on the total time, being at least two orders of magnitude greater that the time required to compile the files into executables. We conducted 10 trials of this experiment, and obtained an average time of 3197.46 sec with a standard deviation of s = 0.95 sec.

3.3. Attack Experiment

The Attack Experiment allowed us to compare the throughputs of both unintended USB channels with the values predicted by our mathematical models, and to confirm the Hardware Trojan Horse Device's processing capability. For 20 trials, the Hardware Trojan Horse exfiltrated a 161 byte file using the Keyboard LED Channel and processed that file to cause the exfiltration of two files totalling 877 bytes using the Audio Channel. The Hardware Trojan Horse successfully processed the file exfiltrated using the Keyboard LED Channel in all instances, properly causing the exfiltration of the correct files using the Audio Channel and opening up the tunnel and back door.

For the *Keyboard LED Channel Throughput*, the mathematical model from Section 2.1.2 predicted that the file transfer would take 70.956 sec, resulting in a throughput of 2.269 bytes/sec. The average exfiltration time was 70.802140 sec (s=0.042964 sec) with a throughput of 2.274 sec. This represents a difference between the predicted and the observed throughput of 0.22%.

In the case of the *Audio Channel Throughput*, the mathematical model from Section 2.2.2 predicted that the files would be transfered in 0.130000 sec, with a throughput of 784.615 kilobytes/sec. Over the course of the 20 trials, the average exfiltration time was 0.129987 sec (s=0.000002 sec) corresponding to a throughput of 784.695 bytes/sec. For this

channel, the difference between the observed and predicted throughput is even smaller at 0.01%.

3.4. Discussion

The results of the Upload Experiment confirmed that it is possible to steal a user's credentials and use them to upload arbitrary code using the keyboard functionality of the Hardware Trojan Horse. The time required to complete the upload of our chosen applications was approximately 57 minutes. While this is slow, we believe that it represents a significant risk because it highlights an avenue of attack that has not been well researched.

The results of the Attack Experiment indicate that the Hardware Trojan Horse can: 1) use the unintended USB channels to exfiltrate data from a network endpoint; 2) process the exfiltrated data and analyze it to decide on further action; and 3) use a tunnel and a back door to further compromise the network endpoint. The throughput of these unintended channels is non-negligible. The Audio Channel is particularly dangerous with a throughput in the hundreds of kilobytes/sec.

The Keyboard LED Channel's observed throughput of 2.274 bytes/sec is 66.40% of the 3.42 bytes/sec Theoretical Throughput discussed in Section 2.1.1. The difference is due to the rudementary coding scheme we chose for the channel. The implemented channel uses two bits of a possible 3-bit information field and has a one byte overhead introduced by framing symbols. We suspect that a more efficient coding scheme could be used to improve the throughput of the Keyboard LED Channel. The Audio Channel's observed throughput of 784.695 kilobytes/sec is 76.71% of the 1023 kilobytes/sec Theoretical Throughput. This difference is due to the use of 1020 byte data packets rather than 1023 byte data packets and the effect of the observed end-stuffng.

The Hardware Trojan Horse's integral processing capability makes it very resilient. Even if the network endpoint is sanitized by reinstalling all its software, it remains vulnerable. The Hardware Trojan Horse will remain able to steal the legitimate user's credentials, upload and execute arbitrary code for as long as it is connected to the network endpoint.

3.4.1. Observability. Someone with direct observation of the network endpoint would be able to notice the uploading of applications, as the text entered by the Hardware Trojan Horse device would appear on the display as text being entered by the keyboard. A user at the network endpoint could also disrupt the uploading of the applications because any characters entered on the legitimate keyboard would be passed to the file containing the uploaded application.

Since the Keyboard LED Channel changes the state of the network endpoint's Human Interface Device, its use could also be noticed by someone using the network endpoint. Such users would notice something if they were using an attached keyboard to input text, because the capitalization and use of the number pad would be affected. The use of the Audio Channel could also be detected by a direct observer, should they be using the network endpoint's attached speakers. The use of the Audio Channel results in a short disruption in the playback to the speakers, and this is noticeable.

The possible observation of the Hardware Trojan Horse device leads us to determine that its application uploading and attack phases should occur outside of the organization's regular business hours. This would greatly reduce the probability that the Hardware Trojan Horse device would be discovered. The activity of the Hardware Trojan Horse would still be recorded by the network endpoint through logs of user activity, but as this avenue of attack has not yet been well explored, we doubt that many organizational policies would look for such activity. Therefore, even if the Hardware Trojan Horse device activity was detected, it would likely be attributed to a human sitting at the network endpoint.

4. Conclusion

The research community does not yet fully understand the risk associated with USB devices. The work presented here investigated a portion of this risk, specifically the risk associated with unintended USB channels. We have proved our hypothesis that the USB protocol carries the risk of unintended USB channels.

The work is important in highlighting deficiencies of current USB device risk mitigation strategies that rely on device self-identification such as Endpoint Security Solutions. A Hardware Trojan Horse device can circumvent such risk mitigation strategies because it represents itself as Human Interface Devices, which are poorly regulated by Endpoint Security Solutions.

The attention that we bring to Hardware Trojan Horse devices is in itself an important contribution of our work. Because such devices have their own processing capabilities, they will not be easily detected by current security policies and practices. It is also possible to envision many different scenarios where a Hardware Trojan Horse device could be used. Physical access can now be potentially sufficient to compromise a network endpoint, without attempting access through the network.

4.1. Future Work

Our work only begins the investigation into the risks associated with unintended USB channels and Hardware Trojan Horse Devices. The following areas deserve more attention.

4.1.1. Investigation of other channels. As demonstrated by the validation of the Keyboard LED and Audio Channels'

Throughput Models, the achievable throughput is less than the Theoretical Throughput. The investigation into the risk of USB devices using unintended USB channels would benefit from determining achievable throughputs for other identified unintended channels. In this way, defenders could more accurately assess the risk associated with USB devices.

We have concentrated on throughput for the characterization of unintended USB channels, and we have briefly discussed the observability as another characteristic of interest. We feel that other characteristics should also be examined. Recent research has demonstrated a link between covertness and throughput of a channel [20]. Should defenders begin to monitor for unintended USB channels, the two channels used in this work could potentially sacrifice achievable throughput in order to become more covert. Many extant lines of research regarding the detection and mitigation of covert channels could be leveraged to develop covert USB channels. Reliability can be increased through the introduction of error correction codes. Such codes would address the deficiency of the Keyboard LED Channel's observed insertion error. Reliable covert USB channels can pose a serious risk to the community.

4.1.2. Physical Implementation. The Proof of Concept Hardware Trojan Horse device developed for this research is not an actual attack tool. The investigation into USB device risk would benefit from actual implementation of a Hardware Trojan Horse device mascarading as a Human Interface Device such as a keyboard. Moreover, actual implementation of a workable Hardware Trojan Horse device would allow for the determination of other important channel attributes, such as the effort required to design and implement.

USB devices are ubiquitous and they present a risk to the security of contemporary computer systems. The results of this work contribute to the research community's understanding of USB risk by discussing unintended USB channels and Hardware Trojan Horse devices. This work clearly demonstrates that the risk of USB devices using unintended USB channels is significant. The research community must now take action to find effective mitigation strategies.

Acknowledgment

This research was funded in part by the ISSNet, a NSERC Strategic Network (http://www.issnet.ca/).

References

[1] USB Implementers Forum, "USB 2.0 Specification," 2001, http://www.usb.org/developers/docs.

[2] S. Stasiukonis, "Social engineering, the USB way," 2006, http://www.darkreading.com/document.asp?doc\id=95556.

[3] ——, "Social-engineering employees," 2007, http://www.darkreading.com/document.asp?doc\id=140433.

[4] CentenialSoftware, "Devicewall home page," 2009, http://www.devicewall.com.

[5] CheckPointSoftware, "Pointsec protector homepage," 2009, http://www.checkpoint.com/products/datasecurity/protector.

[6] J. Clark, "An examination of endpoint security methods to regulate USB flash drives use," Royal Military College of Canada, M.A.Sc. Depth Research Paper, Oct. 2007.

[7] DeviceLockInc., "Devicelock homepage," 2009, http://www.devicelock.com.

[8] J. Axelson, *USB Complete*, 3rd ed. Madison WI, USA: Lakeview Research LLC, 2005.

[9] D. Barral and D. Dewey, ""Plug and Root", the USB Key to the Kingdom," 2005, http://www.blackhat.com/presentations/bh-usa-05/BH\US\05-Barrall-Dewey.pdf.

[10] M. Al-Zarouni, "The reality of risks from consented use of USB devices," in *Proceedings of the 4th Australian Information Security Conference*, Sep. 2006, pp. 5–15.

[11] G. Shah, A. Molina, and M. Blaze, "Keyboards and covert channels," in *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[12] Common Criteria Recognition Agreement, "Common criteria for information technology security evaluation, version 2.3," 2005, http://www.commoncriteriaportal.org/thecc.html.

[13] PLX Technology, "Net2280 home page," 2008, http://www.plxtech.com/products/net2000/net2280.asp.

[14] USB Implementers Forum, "Device Class Definition for Human Interface Devices," 2001, http://www.usb.org/developers/docs.

[15] HHDSoftware, "USB Monitor Profession Homepage," 2009, http://www.hhdsoftware.com/Products/home/usb-monitor-pro.html.

[16] USB Implementers Forum, "USB Device Class Definition for Audio Devices 2.0," 2006, http://www.usb.org/developers/devclass\docs/Audio2.0\final.zip.

[17] Microsoft, "Windows Media: WAVEFORMATEXTENSIBLE," 2008, http://msdn.microsoft.com/en-us/library/aa391547(VS.85).aspx.

[18] ——, "Windows Driver Kit: Audio Devices WaveCyclic Latency," 2009, http://msdn.microsoft.com/en-us/library/ms790342.aspx.

[19] Tarako, "EXE2VBS," 2003, http://www.haxorcitos.com/ficheros.html.

[20] R. Smith and G. Scott Knight, "Predictable design of network-based covert communication systems," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, May 2008, pp. 311–321.