Creating a Linux based wireless HID injector with remote payload interface.

Marc Helbling

Haaga-Helia University of Applied Sciences
Business Information Technology
Thesis
2023

Abstract

Author(s)
Marc Helbling

Degree
Bachelor of Business Administration

Report/Thesis Title
Creating a Linux based wireless HID injector with remote payload interface.

Number of pages and appendix pages
29

USB is a flexible protocol that provides a lot of functions and HID functionality which is trust by default on most host systems. This makes it not only versatile but also an attractive point of attack. Among the wide range of USB based attacks, even simple methods like keystroke injections represent a significant security risk.

In this paper we explore the dangers and security concerns of USB, by developing our own wireless HID keystroke injector. We examine the core workings of the USB protocol and have a closer look at HID based attack vectors.

During our research and development, we found that the security risks of USB based attacks are far greater than we initially assumed. Such attacks are extremely cheap and simple to deploy, even for bespoke solutions. USB focused attacks are nothing new, but the stagnancy in security improvements implies that the dependency on the legacy implementation of the protocol is too strong to allow for significant changes in its core architecture. If and how this issue will ever be addressed in the future remains to be seen.

Key words
USB, Protocol, HID, Keystroke, Injection, Payload

Table of contents

1 Introduction

1.1 Background

HID injection refers to “Human Interface Device” injection. In our case the focus is specifically on the HID injection over the USB port. With the term injection we refer to a specific type of attack which aims to manipulate or control a target host system with the help of a specialized USB device that is directly connected to a USB port of the target system. This device masquerades as one or multiple legitimate USB devices, such as a keyboard or mouse and sends direct input commands without the knowledge of the system owner. Since such USB HID devices are trusted by default they serve as an attractive form of attack. For example, HID injection can be used to place malicious code on a system, execute programs and commands, for data and information theft, and various other actions.

This project was chosen by us since it strongly reflects our personal field of interest and covers some important core concepts of IT security. Additionally, the project offers a terrific opportunity to combine many of the skills acquired during university studies with an interesting hands-on project. We specifically decided to write the code in Golang since it is a very efficient language of growing importance, and to our knowledge nobody seems to have previously developed a solution like ours with Go. A personal bonus is the fact that it will help us to improve our own knowledge of the language. As a fun fact most of the publicly available solutions which are similar in nature are developed in Python.

The advantages achieved with this project are to increase our own knowledge, as well as to create something impactful from commonly available hardware that is extremely cheap and easy to acquire. The project shall be a foundation for us, which serves as the core for a variety of useful tasks in both, penetration testing, as well as educational demonstrations to raise security risk awareness. While we are developing our own solution from the ground up in Golang, we would like to point out that there are various commercially available devices and some open-source projects, which provide a similar approach to keystroke injection of a varying degree.

These solutions share aspects with our project, but we are not aiming to produce a direct clone of an already existing solution, nor are we reinventing the wheel. If you are interested in commercial solutions, some of the most popular products on the market are the Rubber Ducky, and the Bash Bunny. They both rely on a very compact form factor and have a large user community and great collection of preexisting payloads. Nevertheless, there are multiple other products and projects that cater to similar use cases.

1.2 Problem Statement

Security in the field of IT has become an increasingly crucial element (CVE 2023). With a growing number of critical resources and infrastructure being driven by IT, it is an extensive and arduous task to defend against a growing pool of attack vectors. One of those vectors is the category of hardware security. In this paper we will specifically focus on malicious Human Input Devices (HIDs) that connect through the USB (Universal Serial Bus) protocol and inject commands unknown to the owner of the machine while mimicking an input device such as a keyboard.

USB device-based attacks have grown in popularity in recent years and represent a significant danger to potential victims (Nissim, Yahalom & Elovici 2017). The reason HID injection has such a large potential for malicious use stems from flexibility which is inherently forced by the USB protocol and therefore directly impacts the security of most operating systems that support the standard (Nicho & Sabry 2022).

It certainly is not unheard of that some environments take sufficient measures in this regard, may it be by physically disabling USB ports, or using solutions similar to the USB Keystroke Injection Protection tool which was released by Google (GitHub google/ukip, 2 May 2023). Nevertheless, the area itself remains an often-overlooked topic, especially since protections against such attacks can be a major inconvenience for the users. In short, the dangers of HID injection attacks are generally underestimated, and their scope not well enough understood, which directly impacts the safety of various computer systems and IT infrastructures.

1.3 Goals & Objectives

The objective of this thesis is the configuration of a Linux based SBC and the development of an accompanying software solution written in Golang which utilizes the USB HID gadget driver to mimic a USB peripheral and inject HID specific input into the host system. This shall allow for keystroke injection over the USB HID protocol by translating payload inputs and sending them forward as scan codes which are in turn interpreted as legitimate keystrokes by the target host (USB Implementer’s Forum 2004, 53). Communication and Payload delivery will occur with the help of a WebSocket connection and a web interface which is hosted on the SBC itself. One shall be able to connect the SBC to a USB port of a host system, establish a connection with the WNIC and deliver keystroke-based payloads with the help of the web interface.

We additionally aim to acquire a deeper understanding of the USB HID protocol (USB Implementer’s Forum 2001, 1) and the potential techniques that can leverage keystroke injections. Furthermore, we want to explore as much as possible about the limitations and possibilities of such

an attack vector and raise awareness about the possibilities and dangers if a system is not protected against said methods.

Our goal is to provide a fundamental software solution as well as the system configuration to create a HID injection tool with the help of a cheap Linux based SBC that can then be used for the purpose of raising awareness through demonstrations, or in field applications like penetration testing. All the work will be made publicly available on GitHub (GitHub m37/HIDNinja, 2023) and covered under the GNU GPLv3 License (GNU Project, 2023).

We aim to keep improving and extending this tool in the future, and to implement additional features and functions. If you are interested in this project and possess skills that could assist in the implementation and development of additional features, please consider contributing to the development of HIDNinja through GitHub. Any contributions will be highly appreciated.

2 Theoretical Background

2.1 USB HID Protocol

To successfully mimic a USB HID, in our case a keyboard, we must comply with the device class definition for human interface devices (USB Implementer's Forum 2001). Our core foundation is based on the Linux USB HID gadget driver (The Linux Kernel Archives 2023; Linux USB Project 2023). The gadget driver allows us to configure and create a USB gadget and define the properties and functions of an emulated USB device. If wanted we are theoretically also able to change the type of HID device during its operation (Nissim, Yahalom & Elovici 2017, 8) to provide for an even broader spectrum of attack, and evasion techniques. Since the USB protocol is trusted by default in most operating systems (OPSWAT 2014) it will allow us to inject input data as soon as the device we emulate has completed the so-called enumeration process with the target/host system (Axelson 2015, 91-121).

Before we can successfully emulate a device and establish successful communication with the host, we should have a deeper look inside the workings and specifications of the USB protocol to gain a fundamental understanding of the way things are supposed to work, and how we can accurately comply with the protocol standards. Thanks to the flexibility of the USB standard, a single device can also be designated as a composite device and allows for the assignment of multiple functions such as video and audio and the same time (Microsoft 2021). Meaning that we are not limited to a single functionality when masquerading as a USB device.

Every assigned function needs its own endpoint which in turn serves as the communication channel/pipe (Nissim, Yahalom & Elovici 2017, 4). Without the establishment of a pipe there is no data transfer possible. In total there are four distinct types of data transfers available to us that can be used (Axelson 2015, 34). The first type is control transfers for message data, which are responsible for the handling of status responses. A control transfer pipe is a default pipe for all devices and always uses the endpoint zero. The other three pipes are dedicated to stream data. They handle isochronous transfers, interrupt transfers, and bulk transfers. (Axelson 2015, 33-40)

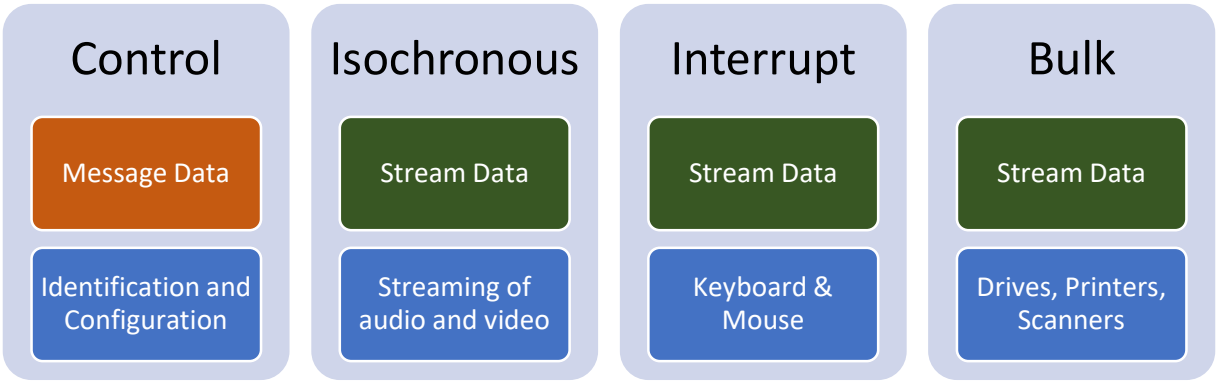

Figure 1. Stream pipe transfer types (Axelson 2015, 39-42).

Once a USB device is connected to the target system (may it be emulated or real) the following procedure occurs. The host detects that a new connection on the ports data line has occurred, and the communication speed is determined by observing the data lines. Once these criteria are satisfied the host will initiate a connection reset and proceed by requesting the device descriptor by sending out a *Get_Device_Descriptor* command that prompts the USB device to send over the descriptor length and descriptor itself. After another connection reset the host assigns a unique address to the USB device with the *Set_Address* command and requests the device configuration by sending out *the Get_Configuration_Descriptor* command. A configuration descriptor can define one or multiple interface descriptors. Each interface descriptor contains >=0 endpoint descriptors, which is crucial when we are dealing with a composite device. For the enumeration process to proceed, the USB device will then need to fulfill the request with the delivery of the descriptor configuration. Please note that this is a simplified representation of the USB enumeration process which ignores some lower-level aspects. (FTDI Chip 2009, 15; The Linux USB sub-system 2023; Axelson 2015, 93-97).

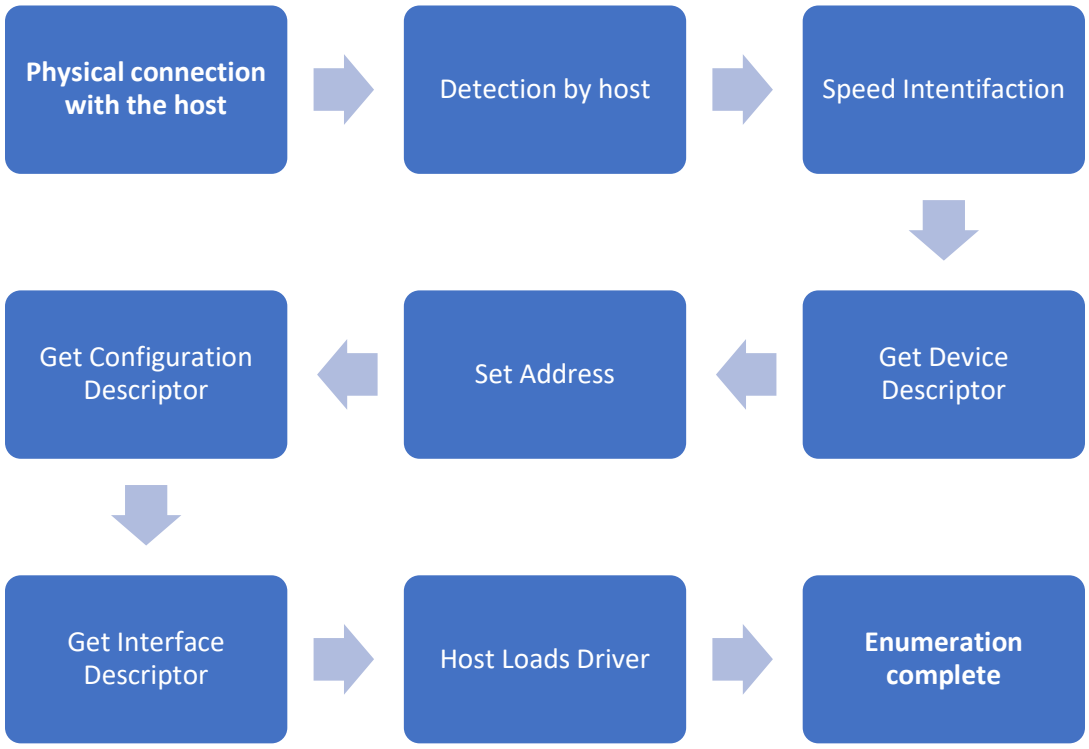

Figure 2. Simplified USB enumeration process flow

Once this identification procedure is successfully completed the host will attempt to acquire or pick a driver that matches the identified device. The driver selection is purely based on the USB classification as well as the vendor and product id that the device provides to the host. With the completion of this step the communication channel between the host and USB device is fully established. With this knowledge in our hands, we now have a better understanding of how the protocol works and what dangers it is open to. (Axelson 2015, 91-121)

2.2 Wireless Communication

While payloads can be stored directly on a physical device, we gain a lot more flexibility if we can manage, send, and distribute payloads remotely. For wireless communication with the SBC one of the most straightforward options is a simple Wi-Fi connection. Alternative methods like Bluetooth could also be applicable depending on the use case scenario. Depending on the communication method and distances involved, external antennas might be required for improved signal strength. While other communication methods such as radio frequencies (RF) are possible with certain hardware adjustments, the base board SBC we are using does not provide any other default means of wireless communication besides Bluetooth 5.0 or Wi-Fi 2.4GHz and 5.0 GHz (Raspberry Pi, 2023).

Nevertheless, methods of communication like RF have proved themselves to be extremely useful for stealthy solutions like the NSA developed COTTONMOUTH-I (Wikipedia 2023). This device was implanted into a USB cable header. It had a wide range of functionalities and was among many things not only able utilize the USB protocol for payload delivery, but could provide data exfiltration, infiltration, data relay, and remote control over RF communication. (Schneier on Security 2014). An RF signal is much less likely to be detected or cause any suspicion compared to an active Bluetooth or Wi-Fi connection, but it comes with its own set of technical challenges.

2.3 Delivery & Execution

Depending on the capability of the implemented software interface we are in complete control of the delivery once a connection to the target system is successfully established. The payload delivery can occur either by preconfigured keystroke inputs that are executed as soon as the SBC is connected to the target system, based on a timer, or they are delivered remotely. This allows for various scenarios of keystroke injections and a flexibility which covers multiple use cases. Naturally, if a system is locked or powered down, our potential of interaction might not be as impressive. (Crenshaw 2011)

When sending payloads to the SBC, the received data needs to be translated and interpreted correctly on the other end (USB Implementer’s Forum 2004). Especially if we consider that a lot of payloads will equate to multi-key sequences and shell commands, we must ensure the correct interpretation of keystrokes. This includes modifier keys such as SHIFT, CTRL, ALT and the like. The translation of most characters is a rather straight forward process, while the modifier keys require a bit more attention so that they are recorded and transmitted correctly from the point of payload design in a text format, up to the transmission, receival, translation and ultimately execution. Luckily something as simple as the utilization of regular expressions (RegEx) should enable us to catch and identify specific patterns or symbols that represent a modifier key within a payload string.

An additional consideration which must be made is speed. The SBC will inject keystroke inputs at an incredible rate. When we are sending multiple commands, we are likely to input them faster than the target host can process them. This can cause issues and potentially hinder successful execution. Therefore, it is essential to implement a method that allows us to instruct delays between commands inside a payload if needed. Keystroke injection protections are another element that could get in our way, since they rely for their heuristics on the timing and speed of keystrokes (Google Open Source Blog 2020; GitHub Google/ukip 2023).

2.4 Attack Vectors & Payloads

When we are thinking about general USB devices, most people might not perceive them as particularly dangerous. While there is a growing awareness about the risk of USB storage devices like thumb drives, there is a certain innocence associated with most other USB peripherals. Figure 3 demonstrates the successful hardware integration of a keystroke injector into a regular mouse, while maintaining full functionality.

Figure 3. Modified Mouse with wireless HID injection capability (Paganini 13 June 2017).

This represents a fantastic opportunity for malicious actors that want to leverage the wide range of functionality that is accommodated by the USB protocol and its trust-by-default environment (Tian & al. 2018). While this paper is heavily focused on HID based injections, it is important to understand that there are various other malicious attacks that can be achieved over USB. While keystrokes and mouse movements are an exceedingly popular attack form, there exist other methods such as the theft of network traffic over emulated USB Ethernet adapters or driver related attacks where the host is prompted to install a malicious driver (Nicho & Sabry 2022, 245).

Returning to the topic of HID injection, we will now have a closer look at some of the possible payload scenarios that could occur with the assistance of malicious keystroke injections. While the following approaches sound straight forward on paper, varying system configurations, user rights,

and other variables, might require us to implement adjustments, workarounds, or employ an entirely different angle of attack for successful payload execution.

The scenarios described by us are based on preexisting payload scenarios from opensource projects as well as commercial products such as the Bash Bunny and Rubber Ducky, which are hosted on repositories such as GitHub and the publicly accessible payload hub from hak5.org (Hak5 2023). Thanks to keystroke injection it is no challenge to place all kinds of malicious scripts and services on a target machine. With most payloads it is wise to obfuscate and hide their operation and existence from the host system as well as possible. The presented examples are just a few of the many existing possibilities for each scenario. Many of these examples overlap with each other and are often combined in a real-world deployment. If needed, you can easily find various other payloads with a quick online search.

2.4.1 Scenario 1: Credential Theft

- Keylogging: Simply capture live keystrokes by creating a persistent service on the target.
- Fake authentication: Create a fake authentication message/popup that prompts the user to enter their credentials.
- Browser Password Dump: Extract saved credentials from a vulnerable browser.
- Session Cookies: Extract session cookies from a browser
- Wi-Fi Passwords: With the help of something like the “nmcli” command on Linux, or “netsh” on Windows, we can easily grab the password of saved Wi-Fi networks.
- Mimikatz: Download and run third party tools like Mimikatz that extract authentication credentials from the system memory.

2.4.2 Scenario 2: Reconnaissance & Data Collection

- Tree: Obtain an overview of target systems file and folder structure with the “tree” command.
- SYSVOL: we can look for service account credentials in the SYSVOL GPP (group policy preferences)
- Network: Try to get a grasp on the network configuration and topology through scanning of the network and viewing of host configurations. Deep scanning of the LAN might create a lot of noise, but you will get a lot of valuable information.
- Processes: Get information on all running processes on the host with the help of bash or PowerShell.
- Shell History: As example we can look at the shell history in a place like “/home/$USER/.bash_history\” on Linux.
- Clipboard: Create a script that records all clipboard contents
- Screenshots: Create a script that takes and gathers screenshots

- Traffic: Use a tool like “tcpdump” to catch the network traffic
- GPO settings: you can pull the GPO settings on Windows with the “gpresults”.
- Passwords: sometimes passwords are stored in files or in the registry, so we can specifically search through those.
- Active Directory: There is a wide range of PowerShell commands that fetch information like the Domain controllers, password policies, user properties, group members, etc.

2.4.3 Scenario 3: Phishing

- DNS Poisoning: Redirect domains by modifying the host file or changing the DNS server.
- Fake commands: By creating a persistent alias we can spoof commands like SSH to execute a spoofed version that grabs the credentials.
- Dialogs: create fake dialogs, as example with “kdialog” on Linux to fix for specific data.

2.4.4 Scenario 4: Execution

- Account: Add a hidden local user/admin account so you can gain access even if the host is logged out.
- AV: Create path exceptions for the antivirus software or terminate it completely.
- Exploits: Take advantage of exploits on the target system. You might be able to make use of privilege escalation or something else useful.
- Grab & Run: Download and execute software or scripts from the internet.
- Imposter: Replace existing software with a modified replacement that conceals additional functions

2.4.5 Scenario 5: Remote Access

- Shells: Open a TCP, UDP or ICMP reverse shell on the host system with PowerShell or Bash.
- Remote Services: Set up remote desktop or remote management services. As example the Windows Remote Management (WinRM) could be enabled for a privileged user account.
- Backdoors: Implement or leverage existing backdoors in the existing infrastructure.

2.5 Exfiltration

Exfiltration plays a crucial role in many types of payloads. While there are various ways to exfiltrate data, we are mostly limited in our options if we would like to keep the process quiet and hard to detect. Delivering payloads with keystroke injections is rather worry free, but the exfiltration requires outgoing communication from the target system, which makes the process much more challenging to conceal without tripping any alarm bells. While exfiltration with the help of USB storage devices or to a storage target that can be reached through the Internet is the easiest, it is

also more challenging (but not impossible) to mask from detection (Koval 19 September 2022). Nevertheless, it will not be a viable option if the target host is air gapped.

If a situation arises where a target host is air gapped or stealth is of the utmost essence, there are some incredibly creative solutions available to us. One of these approaches includes the utilization of unintended USB channels to establish a communication path over. For example, it is possible to use the control transfer channel of a USB keyboard, to transfer data using the 3bit keyboard LED channel coding which is meant to handle the indication of the scroll lock, caps lock or num lock engagement. Due to technical limitations, this form of communication caps out at a theoretical limit of 3.42 bytes/s and is therefore somewhat limited in terms of application. A more efficient method which was discovered by the same researchers uses the USB speaker classification. Due to the isochronous endpoint of a full speed audio device the data transfer speed is much higher caps out at a theoretical limit of 1023 kB/s. (Clark, Leblanc & Knight 2010)

Nevertheless, in most situations less sophisticated methods will suffice, and exfiltration through the internet can be still rather covert (Koval 19 September 2022). Specifically, thanks to the increased use of a handful of common cloud providers and their cloud storage that serves as a nice exfiltration target. Thanks to the popularity of cloud infrastructure, a data transfer to a cloud storage from a provider like Google, Amazon, or Microsoft is in many situations less suspicious than if a transfer would be detected to an external FTP server. Especially if the target or target organization is using the same cloud service provider internally, as is used for the exfiltration procedure.

3 Methodology

3.1 Prerequisites & Hardware Specification

For this project, we utilized a single board computer (SBC) which can run Linux. More specifically, we utilized a Raspberry Pi 4B, running Raspberry Pi OS Lite, since we had easy access to multiple units. A different SBC or Linux distribution should also work, but we cannot guarantee that everything will work out of the box.

Ideally, we would have preferred to use a more compact hardware solution like a Raspberry Pi Zero, but none were available to us at the time due to the ongoing hardware shortage.

Figure 4. Raspberry Pi 4B in a Cooler Master Pi Case 40

The connection over the USB-C port on the Raspberry Pi to a USB A port on the target systems was so far sufficient to ensure a stable power delivery and uninterrupted operation. In case of future instability, one could consider underclocking the CPU to lower the power draw. To be on the safe side we did in fact prematurely underclocked all cores since we are not running any resource intensive operations on the hardware.

If you plan to attach any additional peripherals to the SBC, you should be mindful of the overall power requirements and consider an alternative power source instead of the host USB connection. External devices and additional hardware can add a significant power draw to the system. It is essential that you use a USB cable that supports data transfers, since some cables only support power delivery.

3.2 Setup & Configuration

Before we can plug our SBC into another machine and have it masquerade as a keyboard (or any other USB peripheral) and start sending over keystroke injection payloads, we are required to do some configuration work on our SBC. For this purpose, we prepared a corresponding script which runs the correct configurations and creates a service that upkeeps the necessary configuration state. The full configuration scripts can be found on GitHub (GitHub m37/HIDNinja 2023).

The setup script will execute the following changes on our SBC to turn it into a “keyboard”:

1. Check if dwc2 exists in the boot configuration, if not we append it to the file. This will enable the device tree overlay for the dwc2 module which is required to run the USB gadget mode.
2. Check if dwc2 exists in etc/modules if not we append it to the modules file. This adds the dwc2 module to the kernel module list that is loaded on boot.
3. Setting the path for a secondary script which activates the “HID gadget mode” on the SBC and setting the execution permissions for the script (with chmod +x). This subscript contains the following actions:
    - Loading the libcomposite kernel module, which is essential for the setup of USB gadgets.
    - Create a new directory that represents our USB gadget which we aim to emulate. You may also create multiple different gadgets depending on your requirements and use case.
    - Inside our new directory we set the attributes of our USB gadget. As mentioned earlier in our case it will be a USB keyboard. Feel free to adjust these attributes to your liking. Our predefined attributes are:
        - Vendor ID: We went with 0x1d6b which is the ID associated with the Linux Foundation
        - Product ID: We use 0x0104 which corresponds to a Multifunctional Composite Gadget.
        - Device Version: Set to v.1.0.0
        - USB Version: Set to USB 2.0
        - Device Class: Set to 0x02 which translates to “Common Class”
        - Device Protocol: Set to 0x01 which stands for Interface Association Descriptor (IAD)
    - Next up we define the properties of the gadget configuration.
        - Setting the STRINGS_DIR to “strings/0x409” and creating said directory. The code 0x409 stands for US English, which we will be using for our purposes.

    - We set the serial number of our device. We just used here the hex value of “Hello World!” which equates to fedcba9876543210.
    - The manufacturer value we set to Unit37 which is the name of our personal website.
    - For the product name we used HIDNinja Keyboard, based on the codename of our project.
  - Additionally, we are required to add a so-called function for the USB gadget:
    - We create a new folder for said function and define its path in a variable.
    - The protocol value is set to 1 which stands for a keyboard.
    - Subclass is set to 0 which equates to “no subclass”.
    - The report length is configured as 8 bytes and we set the report descriptor for the function. The report descriptor contains a hard coded array of bytes which are responsible for describing the generated data packets (Frank Zhao 2021).
  - Furthermore, we need to set up the USB configuration by defining and creating the directory path for the configuration. When setting the maximum power consumption in mA we chose 250. Other configuration properties include the directory path for English configuration strings, a configuration string directory, and the setting of the configuration string.
  - Now, we can link the HID function inside the configuration to the symbolic link named hid.usb0 within the config and enable the USB gadget through the writing of the gadget name into the USB Device Controller (UDC) file “sys/class/udc”. We are also required to ensure the correct access permissions for “/dev/hidg0” (777), otherwise access to the HID gadget will be denied.
4. Reload the systemd manager configuration to apply changes.
5. Enable the “usb-gadget” service on boot.
  - This systemd service unit file sets up the "hidninja.service" to run the "HIDNinja" USB Gadget during system startup after the syslog service is ready. It runs with root privileges and is enabled to start when the local file systems are mounted.
6. And as a last step we reboot the system.

In summary, we simply enable the HID gadget mode on the SBC through the modification of configuration files, set up specific scripts, and enable a systemd service. Our subscript will allow the Linux system to act as a keyboard that behaves according to our configurations. Beware that depending on the original configuration state of the used SBC your outcome may vary, and additional adjustments may be required.

3.3 Software Architecture and Design

For our project, we chose to create a WebSocket server to act as the middleware and provide us with real-time communication between a frontend web application (user interface) and the backend which is handling the USB communication to the host system. Therefore, the approach which we are employing is rather straight forward in nature.

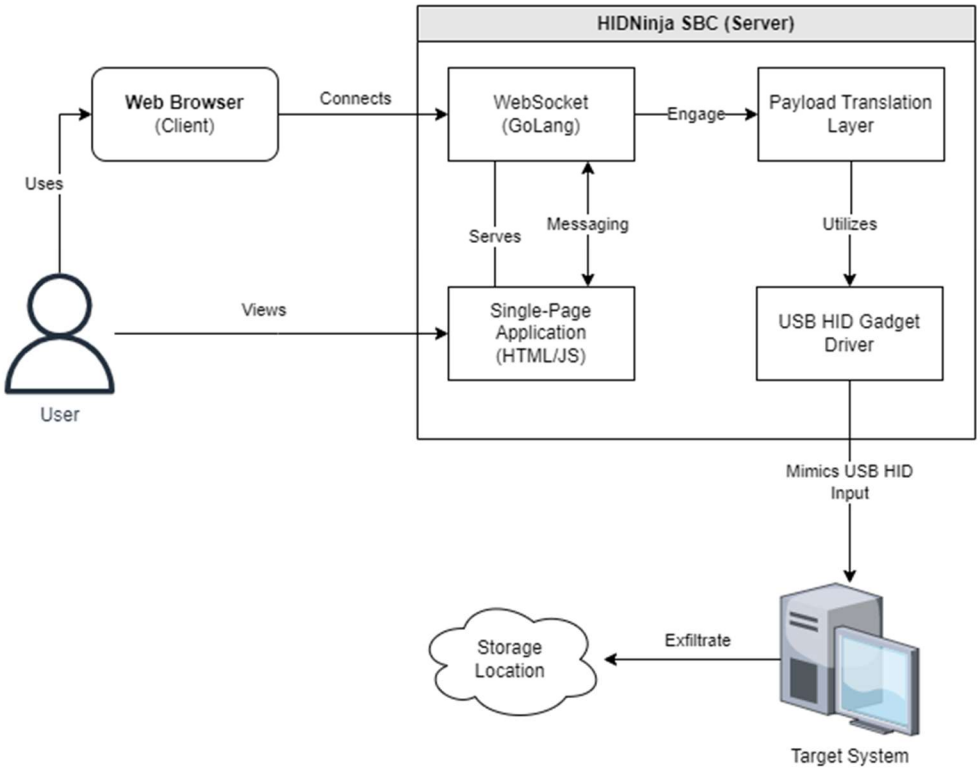

Figure 5. HIDNinja Software Architecture

While figure five provides us with a good representation of the bigger picture, we would like to elaborate on the underlying functionality of our code in some more detail to provide a better understanding of our implementation. We can summarize the core functionality of the software side as follows:

1. The SBC runs a Web Socket Server which sets up routes with a http.HandleFunc() so that we can provide a connection to either the web socket endpoint for payload processing or the web/payload interface. Therefore, the server listens for an incoming client connection and routes it accordingly.
2. If a client connection is detected and the web interface route triggered, the HTML file for the payload web interface is served in return over HTTP.

3. The web interface accepts user input in the form of text-based payloads, which can then be sent back to the web socket servers secondary endpoint route. We implemented a reader function which listens for incoming payloads. Once a payload string is received a status message will be returned to the client and the payload execution process triggered if no errors were reported.
4. Once the payload execution chain is triggered on the server side, the received payload-string is dissected into individual characters and sent to a translation layer. Beware that this process is currently not case-sensitive and will need to be modified in the future to account for modifier keys and other special instructions.
5. The translation layer will run the individual characters through a scan code map where Strings are mapped against specific byte values. The reason why we use string values and not char values here is because we plan to communicate modifiers as multi character strings. This implementation method might be subject to change in future releases. Once the byte value has been correctly identified it will be returned to the payload execution function.
6. With the freshly acquired information we can now assemble a keypress with the scan code byte value. A keypress consists of a byte sequence containing 8 bytes, and each keypress can contain up to six scan codes (USB Implementer's Forum 2001, 30). It is important to remember to also release the keypress again. This can be simply done by sending an zero value byte array, such as []byte{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}.
7. Once the "keypress" is assembled, we pass it to a function that will append the freshly constructed byte sequence to our USB gadget file /dev/hidg0. Once it is appended to the file it will be sent directly as a keystroke to the target host.

The relative simplicity of this design allows us to achieve a somewhat compact and efficient implementation with a conservative amount of code. In the future with added features and functionalities the quantity of code will certainly increase but we aim to avoid code bloat as much as possible.

3.4 Connection & Communication

Since our implementation relies on a Wi-Fi connection, we need to establish said connection to ensure successful communication with our SBC. For this purpose, we open a Wi-Fi hotspot with the name and password matching a set of stored credentials on the SBC. Once the SBC has established a successful connection to the network, we can open a web browser and connect to the payload web interface. If the HIDNinja tool itself is not running yet, you will have to spin it up manually. Once the connection is successfully established and the HIDNinja software is running,

we should be able to reach the web interface and successfully establish the communication. This should be a web address such as: http://raspberrypi.local:3000/. HIDNinja is running a WebSocket on port 3000 (unless configured otherwise) and serves the payload interface once a connection is detected.

The web interface provides a textbox which allows for regular keyboard input. At the press of a button the defined payload string is then sent to the SBC where it is further processed and sent to the target host as keystrokes.

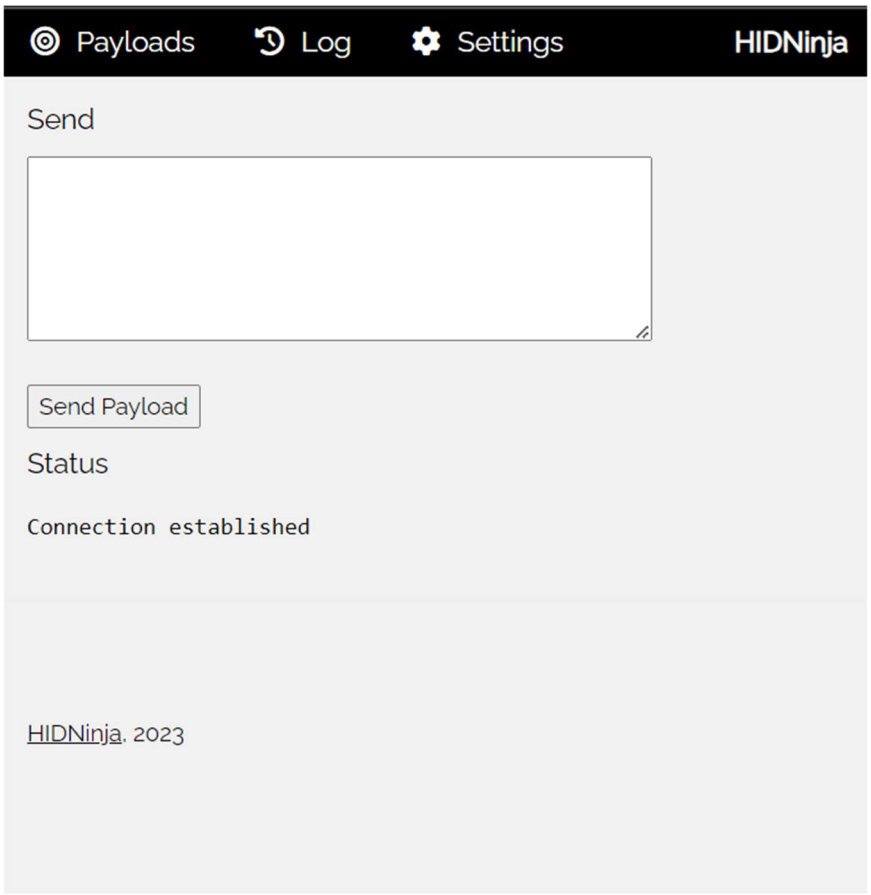

Figure 6. HIDNinja Web Interface

3.5 Payloads

Payloads are entered in the form of plaintext. While basic keys are translated one-to-one, modifier keys require specific formatting so that they can be interpreted correctly by the software. Currently the translation and interpretation of modifier keys is not in the scope of this project. The same applies to a configurable delay between keystrokes. All these elements will be future features that we will implement for the project, since they are only crucial for everything that is beyond the proof-of-concept state of such a solution.

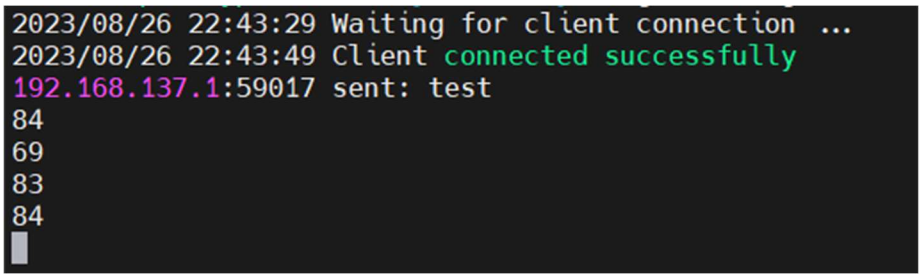

Figure 7. HIDNinja server log of a payload receival and translation into scan codes

3.6 Planned future enhancements

Since the time and resources for this project were strongly limited, the scope is rather barebone in its nature and aims to provide a solid foundation rather than a mature and feature rich solution. Therefore, we have an extensive list of plans for future enhancements and features.

Our currents plans include the following features and enhancements:

- Payload repository and payload management (CRUD)
- Communication support with the SBC for Bluetooth
- Communication support with the SBC for radiofrequency (RF)
- Dynamic USB peripheral emulation (on the fly change of device types) for the purpose of local exfiltration
- Data receival of encoded exfiltration transfers from to Host to SBC through the USB control transfer pipeline to evade detection.
- Dynamic keystroke timing which mimics human input patterns as closely as possible to evade detection algorithms.
- Customizable delays as part of the payload
- Support for modifier keys
- Make the scan code translation case sensitive.
- Support for the handling and managing of multiple SBCs (swarm) from one user interface, and deployment of payloads to multiple targets at the same time.
- Implement relay/mesh networking over multiple SBCs to extend communication range.

4 Prevention and Detection

Keystroke injection attacks present a serious security risk, but there are several methods to detect, counteract and complicate such attacks. Depending on the sophistication of the attack this may be as simple as disabling USB ports or monitoring and analyzing keystroke input patterns and frequencies (Google Open Source Blog 2020). Other common measures are based on more broadly based strategies, they include actions such as the limitation of access and privileges within the operating system itself (Nissim, Yahalom & Elovici 2017) to make it as difficult as possible for a potential attacker to successfully deploy payloads.

While physically disabling or removing the USB ports remains the most efficient way to prevent attacks, it is sometimes an unrealistic option in environments where a USB port is function critical. Removing all doors and windows from a building certainly makes it more secure, but it will not only inconvenience potential thieves but also the owner. If it is possible, we would recommend making USB ports only accessible to specific personnel. As example, disabling external USB ports, and only having an internal USB port inside a locked hardware enclosure.

Even with the consideration that a malicious device must be physically connected to a host to unleash an attack, this can happen faster than one thinks. As previously pointed out, it is essential to remember that malicious USB functionality can be integrated in devices that mimic regular hardware and function as such (Nissim, Yahalom & Elovici 2017). This could for example be mouse, keyboard, webcam, headset, or even a plain USB cable like in the case of COTTONMOUTH-I, which is then strategically planted, installed, or sent to a target of interest.

While we believe that nowadays many users are aware that a random USB stick might present a certain danger. One can easily imagine that most people would not hesitate for a second to plug in some fancy new mouse, keyboard, or headset that showed up at their door. Especially if some social engineering and open-source intelligence (OSINT) is at play, we wouldn’t be surprised if any kind of suspicion is brushed off just too easily. This danger is even more apparent through the increased number of home office workers, where the control over external hardware use is even more challenging. For example, let us consider the following theoretical scenario:

“Mr. Smith works as the head of the claims department of a large health insurance company that handles a lot of sensitive client data. It is known from news articles that the company has introduced three days of home office per week. On the social media channels of Mr. Smith, he frequently talks about his love for gaming and specifically gaming peripherals from Logitech. This is further reinforced by pictures of him that show his private PC, as well as his work laptop with a privately acquired Logitech keyboard and mouse attached to it. A malicious actor could leverage

this information and send over a modified Logitech Gaming product that utilizes USB and attach a letter from Logitech. In the letter one could highlight that the company appreciates his dedication to the brand and wanted to reward a random fan with the best USB headset the brand has to offer.”

We cannot predict for certain if or when Mr. Smith will plug this hardware into his work laptop, but we are confident that the probability of this happening is much more optimal than if he had received a malicious USB stick with an accompanying letter. As is highlighted in this thought experiment, it might not be all that difficult to alleviate any kind of suspicion on the victim’s side thanks to the provided flexibility of USB. Therefore, any USB device should be regarded with the highest caution, especially in environments with critical infrastructure and data. This makes the awareness of such attack vectors one of the most essential elements when it comes to digital security incident prevention.

5 Results and Discussion

With some configuration changes and a bespoke software solution, we were able to successfully utilize a Raspberry Pi 4B as a USB keystroke injector which can be remotely controlled over a web interface. During this project we massively improved our knowledge and understanding of the USB Protocol and its weaknesses that originate from the trust by default nature, as well as its overall flexibility. We learned about multiple weaknesses and attack methods regarding USB which were previously unknown to us. While we initially assumed that keystroke injection by itself is one of the main issues, we quickly discovered that it represents only the tip of the iceberg.

Additionally, we were able to acquire foundational knowledge and experience in the programming language Golang and apply our combined knowledge from fields of IT which were introduced as part of our university degree. Overall, we feel that the topic itself should be creating a lot more noise compared to its current level. Personal conversations with users and IT professionals alike highlighted that the true impact and capability of the USB protocol is not well enough understood by most. It is certainly understandable that not many people enjoy spending their time reading thousands of pages of documentation about USB. But in our opinion, the direct consequence of ignoring the possible impact is rather critical. This is somewhat strange since the security implications that stem from the USB protocols flexibility are nothing new, these problems and the topic has persisted for an extensive period.

We certainly noticed that this is also reflected in the large amount of available information present about the security risks of USB protocol and HID attacks. This knowledge seems to be strongly accumulated in niche areas, such as research papers, selective books, blogs of security researchers, and so on. Therefore, it might be easily overlooked from a more general perspective. For us the big discussion question remains if and how this issue could be addressed from the side of the protocol itself. The open nature of the protocol opens many holes, but the absence of any change implies that they cannot be closed without crippling the functionality or legacy compatibility of the protocol.

6 Conclusion

Malicious USB devices represent a significant risk factor in sensitive environments and critical infrastructures. Without extensive knowledge and training on the topic, most people remain unaware of the potential dangers that lurk within every device that plugs into a USB port. Even without any statistical data on the topic, we can safely assume that a large percentage of users have at some point in their life connected some peripheral over USB to their machine which was from an unknown/untrusted/unverified source or manufacturer. The dangers of USB are not exclusive to keystroke injections and result in a wide range of potential security issues.

As was shown by our own development of a simple HID keystroke injector, the creation of such a device is not rocket science. With some dedication and sufficient research, one can easily develop and tailor a bespoke solution, or just buy it outright from an online shop. The most time intensive element for us was spending multiple weeks of reading through technical specification documents to gain a solid understanding of the USB protocol. Malicious devices that leverage the USB protocol can be created with minimal resources and are readily available on the open market. These conditions by themselves strongly highlight that such forms of attack are not exclusive to state sponsored actors with large funding like the American NSA or the Chinese MSS. They can just as easily be deployed or replicated by smaller actors, groups, or individuals. Therefore, we can safely assume that such attack forms are more common than the general perception would let us believe.

We hope that this paper provides a stronger visibility and better understanding for the dangers of USB in critical environments. May it be for IT professionals or the average users. We strongly hope that projects like ours are a contributing factor for stronger USB security in the future.

Sources

Axelson, J. 2015. USB Complete: The Developer's Guide. 5th ed. Lakeview Research. US.

Crenshaw, A. 2011. Plug and prey: Malicious USB devices. ShmooCon 11. Washington D.C. URL: http://www.irongeek.com/downloads/Malicious%20USB%20Devices.pdf. Accessed: 16 June 2023.

CVE 2023. Metrics. URL: https://www.cve.org/About/Metrics. Accessed: 4 May 2023.

FTDI Chip 2009. Technical Note TN_113: Simplified Description of USB Device Enumeration. Glasgow. URL: https://ftdichip.com/wp-content/uploads/2020/08/TN_113_Simplified-Description-of-USB-Device-Enumeration.pdf. Accessed: 12 June 2023.

GNU Project 2023. The GNU General Public License v3.0. URL: https://www.gnu.org/licenses/gpl-3.0.en.html. Accessed: 10 June 2023.

Clark, J., Leblanc, S. & Knight, S. 2010. Compromise through USB-based Hardware Trojan Horse device. Future Generation Computer Systems, 27, 5, pp. 555-563.

GitHub Google/ukip 2023. USB Keystroke Injection Protection. GitHub. URL: https://github.com/google/ukip. Accessed: 2 May 2023.

GitHub m37/HIDNinja 2023. HIDNinja. URL: https://github.com/mh37/HIDNinja. Accessed: 16 June 2023.

Google Open Source Blog 2020. USB Keystroke Injection Protection. URL: https://opensource.googleblog.com/2020/03/usb-keystroke-injection-protection.html. Accessed: 16 June 2023.

Hak5 2023. Payload Hub. URL: https://hak5.org/blogs/payloads/. Accessed: 16 June 2023.

Koval, K. 19 September 2022. What Is Data Exfiltration? MITRE ATT&CK® Exfiltration Tactic | TA0010. URL: https://socprime.com/blog/what-is-data-exfiltration-mitre-attack/. Accessed: 22 August 2023.

Linux USB Project 2023. Linux-USB Gadget API Framework. URL: http://www.linux-usb.org/gadget/. Accessed: 4 May 2023

Microsoft 2021. How to Register a Composite Device. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/usbcon/register-a-composite-driver. Accessed: 12 June 2023.

Nicho, M. & Sabry, I. 2022. Threat and Vulnerability Modelling of Malicious Human Interface Devices. The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM), 21, pp. 241-247.

Nissim, N., Yahalom, R., & Elovici, Y. 2017. USB-based attacks. Computers & Security, 70, pp. 675-688.

OPSWAT 2023. Detecting and mitigating USB-based threats. URL: https://www.opswat.com/blog/detecting-and-mitigating-usb-based-threats. Accessed: 10 June 2023.

Paganini, P. 13 June 2017. Weaponize a Mouse with WHID Injector for Fun & W00t. Securityaffairs. URL: https://securityaffairs.com/60019/hacking/weaponize-mouse-whid-injector-fun-w00t.html. Accessed: 22 August 2023.

Raspberry Pi 2023. Raspberry Pi 4 Model B specifications. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications. Accessed: 10 June 2023.

Schneier on Security 2014. COTTONMOUTH-I: NSA Exploit of the Day. URL: https://www.schneier.com/blog/archives/2014/03/cottonmouth-i_n.html. Accessed: 12 June 2023.

The Linux Kernel Archives 2023. Linux USB HID gadget driver. URL: https://www.kernel.org/doc/html/latest/usb/gadget_hid.html. Accessed: 4 May 2023.

The Linux USB sub-system 2023. USB Introduction: Enumeration and Device Descriptors. URL: http://www.linux-usb.org/USB-guide/x75.html. Accessed: 12 June 2023.

Tian, J., Scaife, N., Kumar, D., Bailey, M., Bates, A., & Butler, K. 2018. SoK: Plug & pray today–understanding USB insecurity in versions 1 through C. 39th IEEE Symposium on Security and Privacy, SP 2018, pp. 1032-1047.

USB Implementer's Forum 2001. Universal Serial Bus (USB) Device Class Definition for Human Interface Devices (HID). Beaverton, Oregon, U.S. URL: https://www.usb.org/sites/default/files/hid1_11.pdf. Accessed: 4 May 2023.

USB Implementer's Forum 2004. Universal Serial Bus (USB) HID Usage Tables. Beaverton, Oregon, U.S. URL: https://www.usb.org/sites/default/files/documents/hut1_12v2.pdf. Accessed: 6 June 2023.

Wikipedia 2023. ANT catalog. URL: https://en.wikipedia.org/wiki/ANT_catalog. Accessed: 16 June 2023.

Zhao, F. 2021. Tutorial about USB HID Report Descriptors. URL: https://programel.ru/files/Tutorial%20about%20USB%20HID%20Report%20Descriptors%20_%20Eleccelerator.pdf. Accessed: 20 June 2023.