Keynterceptor: Press any key to continue*

Niels van Dijkhuizen, MSc.[1]

Abstract— **The past decade has taught us that there are quite some attacks vectors on USB. These vary from hardware key-logging to driver fuzzing and from power surge injection to network traffic re-routing. In addition to addressing these issues, the security community has also tried to fix some of these. Several defensive hard- and software tools focus on a particular piece of the puzzle. However none, is able to completely mitigate the risks that involves the everyday use of USB in our lives.**

Key stroke injectors like *Rubber Ducky* and *MalDuino* have a big disadvantage: they are not very stealthy. When no protection is in place, there is a big change the end-user will notice something fishy is going on. Proper USB Class filtering policies and a daemon that monitors typing speed will put this kind of attacks to a halt. To bypass both the user's attention and current security mechanisms, I have developed *Keynterceptor*. This is a proof of concept keyboard implant that is able to capture and inject keystrokes and communicate over the air via a back-channel while keeping the local time.

Since Keynterceptor is made up from very affordable, off-the-shelf electronic parts, it is likely that such an attack tool can be created and used by someone with few resources.

I will demonstrate the effectiveness of Keynterceptor in a real-world scenario where an end-point gets compromised.

I. Introduction

In the past 10 years or so, security researchers have presented multiple attacks on the Universal Serial Bus (USB). Section II will cover those techniques in chronological order. With good reason people have claimed that USB is fundamentally broken. Fortunately, the community responded with workarounds to circumvent some of these problems. Section III will cover defensive tools that are freely available at the moment.

In Section IV the idea behind the proof-of-concept *Keynterceptor* will be presented. The subsections will cover the actual implementation and tested scenario.

Finally, the conclusion and possible future work is presented in section V.

II. A decade of USB problems

With physical access to a computer, an easy way to harvest a user's credentials and other sensitive information is to place a keylogger. Often times these keyloggers capture the first x-thousand keystrokes. The oldest reference to a USB based keylogger I could find dates from **2005** (KeyGhost USB Keylogger) [1].

With the introduction of Microsoft Windows 95, the Windows line of Operating Systems include a feature called *autorun* or *autoplay* (depending on the used version of the OS) to automatically start an executable binary from various media. Microsoft research shows [2] that this feature was very popular as propagation method for malware. With Windows 95 up to XP, the default action with CD-ROM drives was to follow autorun.inf instructions without any user interaction. SanDisk and M-Systems developed the U3 technology which added a virtual CD-ROM drive to USB flash drives in order to enable autorun/autoplay-functionality (from now on called autorun). In **2006** HAK5 introduced the *USB Switchblade* [3] [4] to abuse this U3 technology in order to automatically steal LM hashes from systems. Later on, they expanded the Switchblade features with *USB Hacksaw* [5].

Due to the quick spreading of the Conficker worm in 2008/2009, the general public learned about the dangers of the autorun feature. In **2010** Darren Kitchen and Robin Wood (DigiNinja) from HAK5 and Adrian Crenshaw (Irongeek) realised many people therefore disabled autorun. Crenshaw created *Programmable HID USB Keystroke Dongle* (PHUKD) [6] where HID stands for Human Input Device and HAK5 created a commercial product called *Rubber Ducky* [7]. Both are keystroke injection tools that where based on the Teensy development board. When these are inserted in a USB port, they autonomously start typing commands at very high speed for the specified Operating System in order to quickly take control of the machine.

In **2011** Crenshaw combined his earlier research on both PS/2 and USB keyloggers with his PHUKD project in order to capture and inject keystrokes with the same device. He called this project *Hardware Keylogger/PHUKD Hybrid* [8] and as we will see in section IV-A I have continued research in this direction. When I started my project by the end of 2015, I did not know about Crenshaw's work: therefore, I did not use any of his code.

In **2012** Travis Goodspeed presented his *FaceDancer* [9], a device to allow USB devices to be written in host-side Python. This way, fast prototyping and fuzzing of USB device drivers are within reach for many people. An example of USB driver fuzzing and its implications: Skype will crash if it finds unexpected HID descriptors.

In **2014** Karsten Nohl et al. (SRLabs.de) presented a sophisticated attack that reprograms embedded firmware to give existing USB devices a different and covert capabilities. They called the concept *BadUSB* [10] and later that year Adam Caudill and Brandon Wilson implemented exploit code called *Psychson* [11] for the Phison USB 3 controller to further demonstrate the feasibility of BadUSB. Nohl et al. describe scenarios with keystroke injectors (like Rubber Ducky) and rogue network adapters that spoof traffic amongst others.

Later that year Samy Kamkar presented *USBDriveby* [12],

*This work was not supported by any organization

[1]Niels is a lead analyst at the CSIRT of a large organization in the Netherlands [niels.vandijkhuizen@os3.nl].

which is essentially a keystroke injector with HID Mouse emulation.

In **2015** a Russian hacker under the name *Dark Purple* created a *USB Killer* Device [13] [14]. This device charges capacitors via the power lines of a USB port until they are fully charged. It then releases the charge (with a high negative voltage) over the data-lines into the USB port. Since the electrical protection of USB ports is poor, the charge will often reach the CPU, rendering the device useless.

Also in 2015, HAK5 introduced a covert Systems Administration and Penetration Testing tool called the *LAN Turtle* [15]. This is a little computer with the form factor of a generic USB Ethernet Adapter that essentially behaves as a rogue computer in the network. Rob Fuller (Mubix) discovered in **2016** that a Lan Turtle (or USB Armory for that matter) combined with DHCP with WPAD and Laurent Gaffi's *Responder* could harvest credentials from a locked computer [16] [17].

2016 is also the year Samy Kamkar presented his *Poisontap* project [18] which consists of a Raspberry Pi Zero with Node.js. Poisontab presents itself to a locked computer as a low priority network adapter. Through DHCP it tells the targeted machine the local network range is the complete IPv4 space, and therefore overrules the other network interfaces. The browser cache is then poisoned for commonly used domains and cookies are harvested from the locked machine.

Also in 2016, David Kierznowski presented his research on a USB Man in the Middle attack called *BadUSB 2.0* [19]. In his research he uses two FaceDancers: one to emulate a USB Host and one to emulate a USB peripheral. A *Mediating Computer* in between the FaceDancers controls what information gets across the lines. Kierzonwski implementation was able to attack HID-devices.

In **2017** HAK5 introduced its new USB pen-testing platform called *Bash Bunny* [20]. This device can emulate Ethernet, Serial port, Flash storage and HID devices with the help of an automation language. This makes it a versatile USB pen-testing platform.

Finally the Cactus WHID injector developed by Luca Bongiorni and Corey Harding [21] is a WiFi remotely-controlled HID Emulator, based on the popular ESP8266 chip.

A. Wireless HIDs

A related topic on USB security is *MouseJacking* as it is called nowadays. The focus of this type of attacks is at the Radio Frequency communication part of wireless HIDs rather than at the USB side of things. However, it is noteworthy that Max Moser and Philipp Schrödel from remote-exploit.org had already presented wireless keyboard attacks in 2008 [22]. They continued their work and created Keykeriki in 2009 [23] [24]. *GoodFET.nrf* (2011) [25] from Goodspeed and *KeySweeper* (2015) from Kamkar [26] are also able to sniff and inject keystokes from/to some of the more affordable Microsoft wireless keyboards. Finally in 2016, Bastille Research created a Whitepaper describing MouseJacking [27] and shared their code on Github with instructions to create a wireless "Rubber Ducky" [28].

III. Current defences

When we generalise the attacks from Section II the following categories may be used:

- Autorun attacks
- USB Power surge attacks
- USB Keyloggers
- MouseJacking
- Rogue USB devices (BadUSB)

A. Autorun attacks

Since Conficker, autorun attacks - even with U3 technology - have become quite hard (but not impossible) to perform [2]. Good end-point protection (like Anti-virus software) still has its use for known malware. It would be a good practice to use a Scrubber like *CIRCLean* [29] to retrieve safe documents from unknown mass storage devices.

B. USB Power surge attacks

Even though there is not much research in this field, it seems that Power surge attacks (like the USB Killer performs), are defeatable by using an opto-coupler. There are USB 3.0 extenders available, that use fiber-optics to transfer the originally electrical data. These devices are priced around $100,- which is much less than your average laptop or workstation. The extender will break when such an attack is mounted, but since the charge will be stopped at the transit, the computer will survive.

C. USB Keyloggers

The quality of USB Keyloggers may vary. The cheap ones may present themselves with a different USB ID than the original USB device, the better ones operate transparently. The latter category is therefore nearly impossible to detect with software.

D. MouseJacking

It is important to know which wireless technology to use in certain contexts. It would be unwise to use wireless HID devices within highly sensitive environments. On the other hand, if the used wireless protocol is not known for having security issues, one might use it for classed documents up to a certain level. I.e. NIST describes BlueTooth 4.1 BR/EDR, Security Mode 4, Level 4 as a relatively secure communications method [30].

E. Rogue USB devices (BadUSB)

Fortunately there has been quite some research in the field of Rogue USB device detection. Robert Fisk developed a BadUSB firewall called *USG* [31], which will block spontaneous re-enumeration of connected USB devices and it will only allow certain device classes. Unfortunately, if a device presents all sub devices directly without later enumeration, bad intent will not be detected with USG . It currently does not protect against keystroke injection attacks and unfortunately the line speed it too low to be practical for large file transfers.

Dominic Spill developed a USB Man in the Middle tool for affordable ARM-based boards like the BeagleBone Black. The project is called *USBProxy* [32], which is currently able to function as a USB mass storage write-protector. Though a little faster then USG, this solution needs more development in order to function as a generic BadUSB firewall.

Another project that uses the BeagleBone Black is *GoodDOG* by Tony DiCola. His project specifically filters out non mass-storage devices [33].

The *USBGuard* software framework from Daniel Kopeček [34] helps to protect GNU/Linux machines against BadUSB by applying a white- or blacklist based on USB device attributes. It can fingerprint a device by creating a SHA-256 hash from the device descriptor data (stored in base64 encoding). It does not monitor the behaviour of the device.

Another BadUSB defence that specifically targets the Linux kernel, is the *GRSecurity* set of kernel-patches [35] [36]. GRSecurity has the option to:

- Deny new USB connections after activation (toggle)
- Reject all USB devices not connected at boot time

This toggling can be performed from userland. *USBLockout* by xSmurf [37] monitors your user session and triggers this Grsecurity Deny New USB feature when (un)locking a Desktop.

For Windows there is *Beamgun* by Josh Lospinoso [38], which is a daemon that monitors USB device insertions. When a new network adapter or HID devices is inserted, it will disable this new device. Unfortunately, if a device is already known to the system, it will not be able to block it.

Another Windows based solution is *Duckhunt* from Pedro M. Sosa [39]. This tool monitors for inhumanly fast typing by HID devices and blocks the responsible device if this behaviour is detected.

IV. Keynterceptor

After having described important developments on both the attack and defence side of USB technology, in this section will I present some new insights and describe the actual implementation of a keyboard implant. This technique and Proof of Concept implementation will be called *Keynterceptor*.

A. The idea

Key stroke injectors like *Rubber Ducky* and *MalDuino* have a big disadvantage: they are not very stealthy. Even when such a device initially presents itself as a mass storage device only to add an additional keyboard at some point in the future, it has no means to know if the user is still actively working at that computer. When the computer has no USB defences in place, there's a big chance the end-user will notice something fishy is going on. Proper USB Class filtering policies (like USBGuard) and a daemon that monitors typing speed (like Duckhunt) will put this kind of attacks to a halt.

To avoid user attention I thought of a keyboard implant that combines the strengths of a keylogger and a key stroke injector. This way the interceptor is out of sight (either typing on the keyboard or in-line with the keyboard underneath a desk). As long as a user is typing on the keyboard, we 'know' he or she is still there. And while the implant captures keystrokes, credentials can be collected. When a Real Time Clock (RTC from now on) is added to the implant, we can program it to inject payloads with the captured credentials only outside of the target's regular computing time.

Since a keyboard is a commonly trusted device that is not disconnected often, it will be present at boot time - bypassing protections in the line of the GRSecurity patches. To bypass USB device filters like USBGuard, one simply has to clone the USB descriptors of the targeted keyboard. In large organizations, specific brands and models of keyboards are predominant. This makes it feasible to modify such a keyboard model or clone its descriptors. In order to stay under the radar of monitoring daemons like Duckhunt, the keystroke injection routine of the implant should emulate user typing. This can be realized in a rather simple way. Only when a monitoring daemon profiles the typing behaviour of the end-user (called keystroke dynamics within the field of behavioural biometrics), this evasion might fail.

B. The implementation

Having outlined the most important features (i.e. being covert), we need to define some design goals. The PoC needs to be:

- *Affordable* - In order to proof a low level of adversary effort;
- *Autonomous* - Once deployed, it should be able to reach its target without further intervention;
- *Small form factor* - It should be feasible to actually embed the implant in a keyboard;
- *RF Backchannel communication* - Though losing some stealthiness, this greatly enhances attack scenarios.

To illustrate two of the requirements: Kierznowski's BadUSB 2.0 needs a Mediating Computer, therefore it cannot both have a small form factor and be autonomous.

To speed up development, I chose to prototype with existing off-the-shelf modules instead of designing my own printed circuit board. The hardware schematics are shown in Figure 1. For the core of the project I used the 8 bit Teensy 2.0 because of its small form factor and Arduino IDE support. In order to interface over SPI with a MAX3421E USB Host controller the Teensy had to be modified to operate at 3.3V. To achieve this the teensy also has to run half speed (8MHz).

To keep track of the local time, I used a battery-backed DS3231 Real Time Clock chip that contains a very accurate crystal. It communicates with the Teensy over I^2C as does the 24LC256 (256Kb) EEPROM, which is used for configuration storage. An HC-12 module was used for optional backchannel communication. This RF transceiver module operates at 433Mhz and interfaces with the Teensy via the UART/TTL interface.

Apart from the default Arduino/Teensy libraries, I used the USB Host Shield 2.0 library from Circuits@Home [40] and the Arduino DS3232RTC Library v1.0 from Jack Christensen [41].

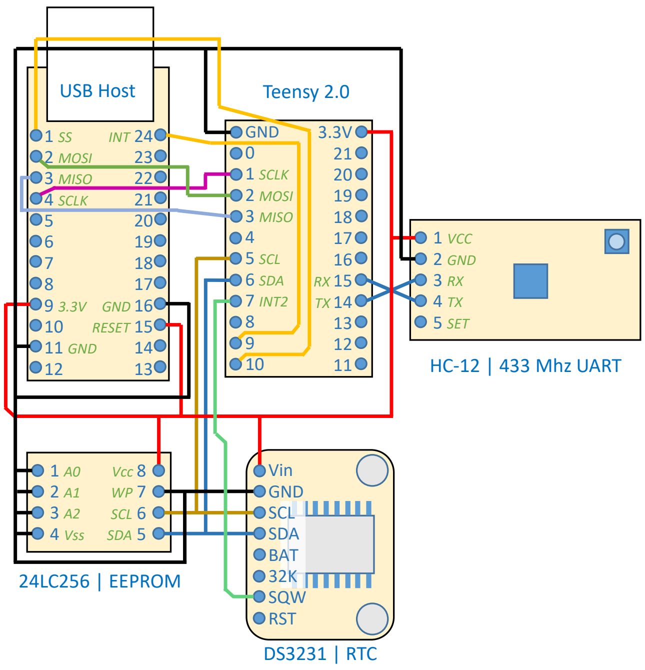


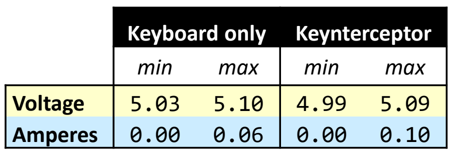
Fig. 1. The hardware schematics of the Keynterceptor PoC

I created a new USB device composition based on the Teensy's USB_HID one (shown in the Arduino environment as: "Keyboard + Mouse + Joystick"). After removing all mouse and joystick classes from all of the files, I modified usb.c and usc_private.h to reflect the correct USB descriptors. In my case the ones from an HP Elite USB Keyboard manufactured by Chicony Electronics. In order to clone the USB table- and report descriptors, one can use the `lsusb -v -d $vid:$pid` and `usbhid-dump -m $vid:$pid` command under GNU/Linux respectively. I also modified `usb_keyboard_class::send_now()` to add some random delays to emulate human typing:

```
int rdelay = rand() % 111;
rdelay += 8;
delay(rdelay);
```

and added `usb_keyboard_class::send_now_quick()` without the delay, for the use of keystroke forwarding.

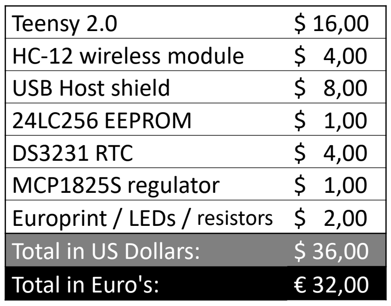
The serial communication speed of the HC-12 is set to 38400bps [8 data bits, no parity, 1 stop bit] for this project. 19200bps would probably have been fast enough, but the effective transmission speed over the air is equal (58000bps with -107dBm receiving sensitivity). When choosing higher speeds, the effective communication distance gets smaller. In order to have a good balance between power consumption, communication distance and speed, the full speed mode (FU3) was chosen in combination with a transmitting power of 17 dBm (50mW) instead of the 20 dBm (100mW) maximum. Keystrokes are sent with 3 bytes per pressed key (1: the keyModifier value, 2: the key value, 3: a line-feed).



	Keyboard only		Keynterceptor	
	min	*max*	*min*	*max*
Voltage	5.03	5.10	4.99	5.09
Amperes	0.00	0.06	0.00	0.10

Fig. 2. The measured voltage and current extremes of the keyboard, with and without Keynterceptor. The voltages and currents have been measured separately.

Since the initial USB 1 and 2.0 specifications describe two types of power sources for powering connected devices, Keynterceptor has to stay within those boundaries. *Low Power* devices get 5V 100mA and *High Power* devices get 5V 500mA. The USB descriptor of the keyboard I cloned describes a low power profile, therefore the total power consumption was measured. Fortunately with the described implementation the difference between regular keyboard use and keyboard use with Keynterceptor is small. The drawn current does not exceed the 100mA limit. Please see the measured values in Figure 2.



Teensy 2.0	$ 16,00
HC-12 wireless module	$ 4,00
USB Host shield	$ 8,00
24LC256 EEPROM	$ 1,00
DS3231 RTC	$ 4,00
MCP1825S regulator	$ 1,00
Europrint / LEDs / resistors	$ 2,00
Total in US Dollars:	$ 36,00
Total in Euro's:	€ 32,00

Fig. 3. The total costs of the Keynterceptor PoC.

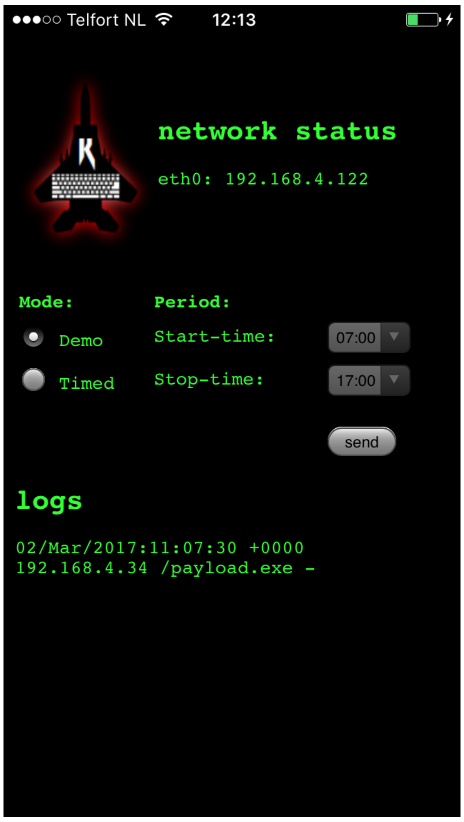
The total cost of the Keynterceptor Proof of Concept lies around 30 euros. The Teensy is the most expensive part of the design. When one would mass produce a new PCB that contains an ATMega32U4, a MAX3421E and a DS3232RTC with pin headers for a RF communication module like the HC-12, the costs will be much lower. Since no other pin-connectors are needed, the resulting device could be made sufficiently small to function as an implant.

C. The tested scenario

With the hard- and software described in Sections IV-A and IV-B, there are some scenarios that could be run:

- Remotely control the targeted keyboard over the air
- Autologin with captured credentials
- Inject keystrokes after inactivity
- Block user input with a RF kill-switch (i.e. for a take-down)

However, to demonstrate some practical use of the Keynterceptor, I thought of a somewhat more complex scenario that combines most of the above. In the first place, we add a Keynterceptor *companion* to the scenario that delivers a *malicious* payload to the targeted computer. I chose the companion to be an 8 dollar NanoPi Neo that runs Ubuntu Linux. This tiny computer, together with a UMTS/4G dongle and an HC-12 module is mounted inside an emptied laptop power adapter. Instead of a regular barrel connector, this "power adapter" has a RJ45 connector at one end.



eth0: 192.168.4.122

Mode: Demo [✓] / Timed

Period: Start-time: 07:00 Stop-time: 17:00

```
02/Mar/2017:11:07:30 +0000
192.168.4.34 /payload.exe -
```

Fig. 4. A screenshot of the companion's web-application

The complete scenario is illustrated in Figure 5 and goes as follows:

1) The attacker armed with a smartphone and small backpack filled with a Keynterceptor + companion goes to a location of choice - say, an office.
2) When there is an opportunity to reach an empty room, the companion can be installed nearby a desk with other power adapters on the ground.
3) Once the companion is up and running, it will connect over the 4G link with an OpenVPN server.
4) Shortly after installing the companion, the attacker is able to connect to it trough the OpenVPN tunnel on the smartphone.
5) The web-application of the companion shows the current state of its eth0 interface.
6) The attacker can try various ethernet patches in the room, until the companion gets an IP address.
7) Once it has an IP, the attacker can select the *out-of-office* time range, in which the Keynterceptor may become active. See Figure 4 for a screenshot of the application.
8) Now the attacker plugs the Keynterceptor in between a workstation and a keyboard of choice.
9) The Keynterceptor waits for its configuration by listening on the HC-12 RF module.
10) Back at the smartphone, the attacker presses *send* to transmit the companion's IP address, operation mode, and activation time over the 433MHz back-channel to the Keynterceptor.
11) At this point the Keynterceptor is armed and ready to capture user credentials, the attacker is done at the office.
12) Elsewhere, the attacker can connect to the VPN server, and log into the companion over SSH.
13) All keyboard activities from the user, can now be monitored. Or in case of a more stealthy campaign, the attacker can be notified when the target machine is compromised.
14) In the night, when everyone in the office has left the building and no keyboard activity is registered for a long time, Keynterceptor logs into the computer by itself with the captured credentials - if the workstation is still on of course.
15) It starts a Powershell and retrieves a payload from the companion that resides in the same LAN.
16) The targeted machine is compromised and controllable via the companion (i.e. via a post-exploitation framework like Empire).

V. Conclusion

With this paper and corresponding presentation, I claim to have built a device that is able to overcome the described problems that classical keystroke injection tools have. I tested the Keynterceptor against a proper USB device filter (USBGuard), a keystroke injection monitor (Duckhunt) and a USB Device Firewall (USG) and was able to bypass their mechanisms.

The presented solution is indeed affordable. It can operate on its own and communicate with the outside world over a RF backchannel. It does not consume too much power, and with a little effort it can be produced with a small form factor to function as an implant.

Together with the companion and less than 1000 lines of C, Python and Perl code, I was able to develop and run a complete attack scenario (including a reverse shell / post-exploitation tool). With this scenario I also presented a real-world use-case for such a device.

I can think of three solutions that effectively stop Keynterceptor:

1) Have two- or multi-factor authentication next to a password with each computer-unlock action.
2) Present the user with an additional captcha or a "click the correct picture" challenge to unlock a machine.
3) Use of secure keyboards: one that accepts a client-certificate and does mutual authentication and encryption of typed data.

With this little research project, I would like to increase the ongoing effort to raise awareness on the bad state of USB and the inherent trust we have in our peripherals.

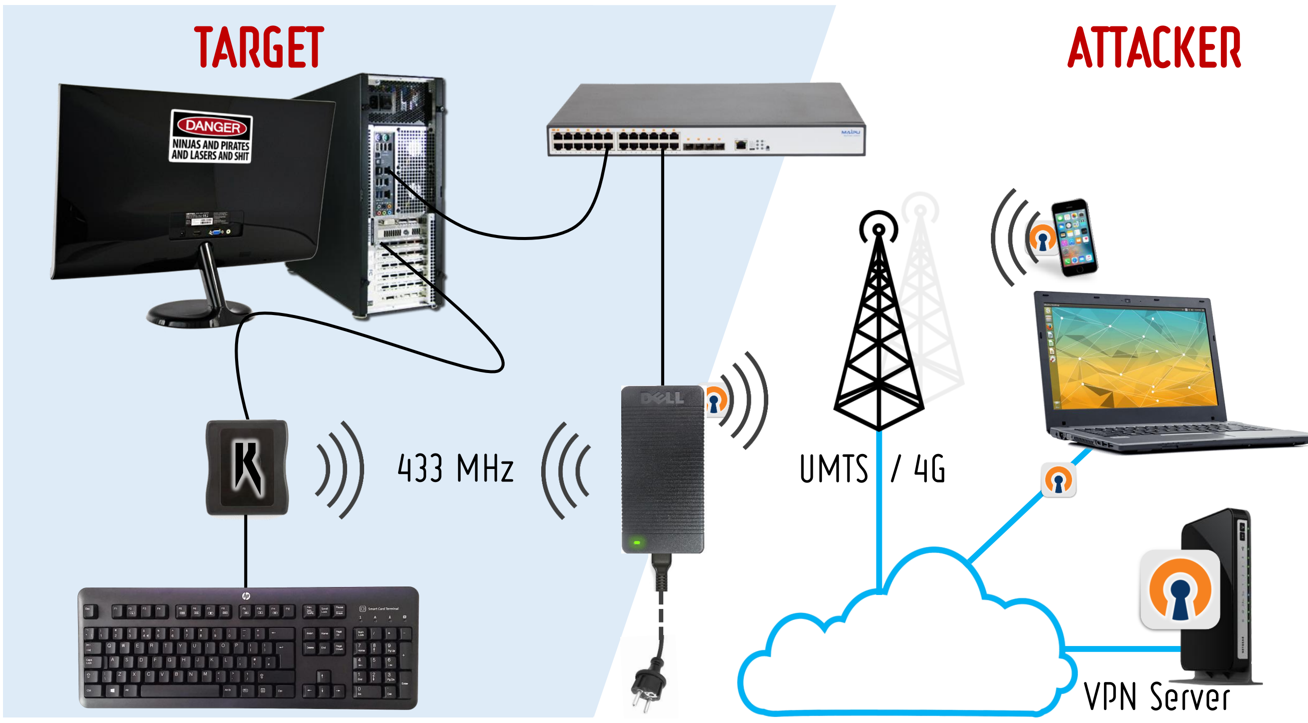


Fig. 5. The complete scenario with Keynterceptor and its companion

A. *Hindsight and Future work*

In hindsight, I realized the external EEPROM was an overkill for the couple of bytes I had to store on it. The Teensy 2.0 already has an EEPROM with 1KB of storage.

I have not taken the effort to secure the backchannel communication, even though the 433MHz UART communication is a bit obscure, proper encryption would still be necessary for secure operations. Another area of improvement might be in the way the USB descriptor cloning is done. With this project, one has to manually clone the descriptors and then recompile the project. It would be more user-friendly to dynamically adjust the descriptors based on what is connected to the Keynterceptor's client-side. I would say this is at least possible with an FPGA, but further research might show whether this could be achieved with a micro-controller as well.

References

[1] "KeyGhost USB keylogger." https://web.archive.org/web/20050205233057/http://www.keyghost.com/USB-Keylogger.htm.

[2] "Microsoft Security Intelligence Report vol. 11." http://download.microsoft.com/download/0/3/3/0331766E-3FC4-44E5-B1CA-2BDEB58211B8/Microsoft_Security_Intelligence_Report_volume_11_English.pdf.

[3] "HAK5's Switchblade." https://web.archive.org/web/20070701192836/http://wiki.hak5.org/wiki/USB_Switchblade.

[4] "HAK5 Forum - Switchblade development." https://forums.hak5.org/index.php?/topic/2361-usb-switchblade-development/.

[5] "HAK5's USB Hacksaw." https://web.archive.org/web/20070514050322/http://wiki.hak5.org:80/wiki/USB_Hacksaw.

[6] "Adrian Crenshaw's HUKD." http://www.irongeek.com/i.php?page=security/programmable-hid-usb-keystroke-dongle.

[7] "Presentation of HAK5's USB Rubber Ducky." http://www.hak5.org/episodes/episode-709.

[8] "Adrian Crenshaw's Hardware Keylogger/PHUKD Hybrid." http://www.irongeek.com/i.php?page=security/homemade-hardware-keylogger-phukd.

[9] "Emulating USB Devices with Python - Travis Goodspeed." http://travisgoodspeed.blogspot.nl/2012/07/emulating-usb-devices-with-python.html.

[10] "BadUSB - On accessories that turn evil." https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf.

[11] "Psychson - Custom Firmware repository on Github." https://github.com/brandonlw/Psychson.

[12] "Samy Kamkar's USBDriveby." http://samy.pl/usbdriveby/.

[13] "The original Russian USB Killer post by @Dark_purple." https://habrahabr.ru/post/251451/.

[14] "The translated USB Killer post." https://kukuruku.co/post/usb-killer/.

[15] "HAK5's Lan Turtle." https://hakshop.com/products/lan-turtle.

[16] "Snagging creds from locked machines - Rob Fuller." https://room362.com/post/2016/snagging-creds-from-locked-machines/.

[17] "Introducing Responder 1.0 - Laurent Gaffié." https://www.trustwave.com/Resources/SpiderLabs-Blog/Introducing-Responder-1-0/.

[18] "Samy Kamkar's Poisontap." https://samy.pl/poisontap/.

[19] "BadUSB 2.0: USB man in the middle attacks - David Kierznowsk." https://www.royalholloway.ac.uk/isg/documents/pdf/technicalreports/2016/rhul-isg-2016-7-david-kierznowski.pdf.

[20] "HAK5's Bash Bunny." https://hakshop.com/products/bash-bunny.

[21] "whid-injector/WHID." https://github.com/whid-injector/WHID.

[22] "27Mhz Wireless Keyboard Analysis Report - Max Moser & Philipp Schrödel." https://www.blackhat.com/presentations/bh-dc-08/Moser/Whitepaper/bh-dc-08-moser-WP.pdf.

[23] "KeyKeriki v1.0 - 27MHz." http://www.remote-exploit.org/articles/keykeriki_v1_0_-_27mhz/index.html.
[24] "KeyKeriki v2.0 2.4GHz." http://www.remote-exploit.org/articles/keykeriki_v2_0__8211_2_4ghz/index.html.
[25] "goodfet.nrf." http://goodfet.sourceforge.net/clients/goodfetnrf/.
[26] "Samy Kamkar's Keysweeper." http://samy.pl/keysweeper/.
[27] "Bastille's MouseJack website." http://www.mousejack.com/.
[28] "MouseJack device discovery and research tools - repository on Github." https://github.com/BastilleResearch/mousejack.
[29] "CIRCLean - USB key sanitizer." https://www.circl.lu/projects/CIRCLean/.
[30] "NIST Special Publication 800-121 Rev.2 - Guide to Bluetooth Security." http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-121r2.pdf.
[31] "Robert Fisk's USG repository on Github." https://github.com/robertfisk/USG/wiki.
[32] "Dominic Spill's USB Proxy on Github." https://github.com/dominicgs/USBProxy.
[33] "Tony DiCola's GoodDog repository on Github." https://github.com/tdicola/GoodDOG.
[34] "Daniel Kopešek's USBGuard website." https://dkopecek.github.io/usbguard/.
[35] "Grsecurity's website." https://www.grsecurity.net.
[36] "Gentoo Linux wiki describing grsecurity patches on USB security." https://wiki.gentoo.org/wiki/Allow_only_known_usb_devices.
[37] "USBlockout subproject of Subgraph." https://github.com/subgraph/usblockout.
[38] "Josh Lospinoso's beamgun repository on Github." https://github.com/JLospinoso/beamgun.
[39] "Pedro M. Sosa's Duckhunt repository on Github." https://github.com/pmsosa/duckhunt/.
[40] "Oleg Mazurov's USB Host Shield 2.0 repository on Github." https://github.com/felis/USB_Host_Shield_2.0/.
[41] "Jack Christensen's DS3232RTC repository on Github." https://github.com/JChristensen/DS3232RTC.