TALLINN UNIVERSITY OF TECHNOLOGY
Institute of Cybernetics

ITI70LT

Piret Lattikas 111557IVCMM

Test automation for web applications with authentication

Master thesis

Supervisor: Andres Ojamaa
Supervisor's degree: M.Sc.
Supervisor's position: Researcher

Tallinn 2014

Declaration

I hereby declare that I am the sole author of this thesis. The work is original and has not been submitted for any degree or diploma at any other University. I further declare that material obtained from other sources have been fully acknowledged in the thesis.

…… ……

(Date) (Author's signature)

TEST AUTOMATION FOR WEB APPLICATIONS WITH AUTHENTICATION

Abstract

The aim of this thesis is to analyze the common authentication schemes used in web applications and the possibility of automating the use of these methods in tests with *Selenium*. Another goal of this thesis is to offer an overview to which extent test automation for web applications with authentication can be achieved with Selenium and how it can be implemented. In addition attention is drawn to the limitations of the Selenium framework in this specific application area. This thesis concentrates on analyzing *HTTP Basic*, user certificate, login form, Estonian ID-card and Estonian mobile-ID based authentication methods.

As an outcome of this work a documentation is presented on how to implement test automation for web applications with these authentication schemes. In addition code examples have been added, which demonstrate how this can be achieved. An analysis of advantages and disadvantages of the different authentication schemes and their usages is provided. The thesis draws attention to the deficiency in the Selenium framework regarding interaction with the browsers security popups. It is concluded that Estonian mobile-ID based authentication with its two factor authentication offers good security while supporting full test automation with Selenium.

AUTENTIMISEGA VEEBIRAKENDUSTE TESTIDE AUTOMATISEERIMINE.

Annotatsioon

Käesoleva magistritöö eesmärk on analüüsida enamlevinud veebirakenduste autentimisskeeme ja võimalust automatiseerida nende kasutamist *Selenium* testides. Lisaks sellele on antud lõputöö eesmärk pakkuda ülevaadet sellest, mil määral on võimalik autentimisega veebirakenduste teste Seleniumiga automatiseerida ning kuidas seda realiseerida. Peale selle pööratakse tähelepanu Seleniumi raamistikus olevatele puudujääkidele antud valdkonnas. Selles magistritöös keskendutakse *HTTP Basic*, kasutaja sertifikaadi, sisenemise vormi, Eesti ID-kaardi ja Eesti mobiil-ID põhise autentimise analüüsimisele.

Magistritöö tulemusena esitletakse dokumentatsiooni ja koodinäiteid sellest, kuidas realiseerida käsitletud autentimismeetodeid kasutavate veebirakenduste testide automatiseerimist. Peale selle on tehtud erinevate autentimisskeemide ja nende kasutamise eeliste ja puuduste analüüs. Lõputöö juhib tähelepanu Seleniumi raamistikus olevale puudujäägile, milleks on võimalus suhelda brauseri turva hüpikmenüüga. Töö tulemusena on järeldatud, et Eesti mobiil-ID põhine autentimisskeem koos kahe osalise autentimisega on turvaline ning toetab hästi testide täielikku automatiseerimist Seleniumiga.

Table of Contents

List of Figures

Introduction

The society is increasingly depending on the Internet and different web applications when handling everyday problems and situations. According to the Statistical Office of Estonia the number of online shopping users in Estonia from the beginning of 2005 until July 2012 has increased almost three times [28]. Furthermore, almost 94 percent of domestic Internet shopping payments were made directly from bank accounts using the Banklink service. Increasing usage of web applications and growing risks, forces application developers to put more effort and thought into web application security. Users of Internet banking have the expectation that their data and transactions are securely processed and unreachable to third parties.

In order to ensure that the web application user is able to see only the data he has the rights to see, the system has to know which user is accessing the application. So to be able to authorize the user, giving him corresponding rights to access the system, it is first needed to identify and then authenticate him. User identity is usually found based on username, personal identification code, user email address or some other similar information. The information obtained during identification is, who the user claims to be [49]. After it is known who the user acts as, it is necessary to verify that the user actually is who he claims to be. Nowadays several authentication schemes can be used for user's identity verification.

Web application security and authentication has to be thought through and included already in the process of writing the requirements for a new information system. These requirements should then be followed in the phase of developing the system. While testing the web application it is also important to test the authentication to ensure its operating reliability. During the functional testing phase it is verified whether the developed module or system behaves in a way that is expected from it [13]. Therefore, as the authentication can be considered a functionality of the application it should be also tested during the functional testing phase.

Many companies use agile methods for development, where short sprints are used, which leaves little time for testing. But developers need to be able to verify their work after

changes and at the same time ensure everything else still works as intended. To enable fast and easy testing for developers test automation should be used. Automated test can be repeatedly run in the same environment and under same conditions to verify information systems quality. Moreover, for web applications there is a need to create automated tests also for graphical user interface. A popular tool for this is *Selenium* [40]. The main purpose of Selenium is automating web browsers for testing purposes. In the scope of this thesis the *Selenium WebDriver* component with *JUnit* testing framework will be used for composing the automated tests. A brief introduction to tools used in this thesis will be given in Chapter 2.

The main goal of this thesis is to analyze the common authentication schemes used in web applications and the possibility of automating the use of these in tests with Selenium. Another goal is to offer a documentation regarding implementation of test automation for applications with authentication with Selenium, because the existing documentation is lacking in this area. This thesis will concentrate on the following authentication schemes: *HTTP Basic*, login form based authentication, user certificate, Estonian ID-card and Estonian Mobile-ID.

The main motivation of this work is the fact there was a lack of documentation concerning test automation for web applications with different authentication schemes with Selenium. In my work as a web application developer I had the actual need for this kind of documentation and analysis. There was a need to write automated tests for a web application graphical user interface that was running in development server, which was protected with user certificate based authentication. Having problems finding a comprehensive documentation about that I started to also think about automating tests for other web applications with different authentication schemes and how can these be done.

In this thesis it is analyzed which authentication schemes testing can be automated with Selenium and shown how exactly it can be achieved. In addition, a side goal is to draw attention to the technical limitations of Selenium framework that hinder test automation of web applications using various authentication methods. There seems to be a great interest towards this subject among people engaged in web application test automation. The thesis is an attempt to provide documentation for testers who come across the same problem.

As a result of this thesis I will present example implementations of automated test classes for web applications with all different authentication schemes focused on in the scope of this thesis. In addition, documentation is provided with steps how to implement these automated tests and what preparations are needed for each. This thesis gives an overview of this topic and should be a good starting point for people who need to get started with test automation for web applications with authentication.

The rest of the thesis is divided into four chapters. Firstly, in Chapter 1 the connection between test automation and cyber security is pointed out and the importance of automated tests to ensure web application security is presented. Secondly, Chapter 2 gives an introduction to the tools used along with Selenium. Moreover, the question of which tests Selenium is a perfect tool for and which tests should not be automated using it is discussed in Chapter 2. Thirdly, in Chapter 3 a closer look is taken at the authentication schemes discussed in this thesis. The advantages and disadvantages of these schemes is pointed out. In Chapter 4, the focus is on test automation for web applications with authentication schemes with Selenium. An answer is given whether automating a particular method with Selenium is possible at all. In addition, code examples of implementing authentication schemes test automation with Selenium is presented. Aspects that need improvement in Selenium are highlighted.

1. Test automation and cyber security

Testing is a process of planning, preparation, and measuring aimed at establishing the characteristics of an information system and demonstrating the difference between the actual and the required status [24]. Sufficient testing of the web application allows the system developers to improve the quality of the information system. Properties such as reliability and security can be seen as parts of software quality and hence gain from the testing process. However, no application can be fully tested and for this reason the critical parts of the system should get more attention during the testing phase.

Automatic software testing can significantly increase the testing that can be done in limited time, so that tests that would otherwise take hours to run manually, can after automation be run in minutes [15]. In addition, a great advantage of automated testing is that the same properties are always tested exactly the same way [26]. Testing in the same environment and in the same state is important to ensure the accuracy and the comparability of the test results.

Given the importance of web applications to their users, the systems have to ensure the protection of confidential and valuable data. The developers of web applications share the responsibility of securing their products. A good way to avoid many security pitfalls is through test automation and verifying the test results after changes. That way it is possible to check that no change presents a security hole in places, which are covered with automated tests. The authentication is a critical part of web applications and test automation for it is therefore important. If these steps are bypassed during testing phase security holes and vulnerabilities may occur in the application later that could have been discovered during the testing phase.

Sometimes it is necessary to add additional security related features after the initial security audit. With this sort of patching some new vulnerabilities may be introduced. Some of these mistakes can be discovered early on with test automation. Thinking through different aspects and covering important places with automated tests can improve web application security and good test result reports are helpful to ensure that to clients and users. Furthermore, if the developer has positive test results he can be more confident

about the security of the development results. It should be emphasized that although automated testing is a valuable approach for improving software quality, when it comes to security, no amount of automated testing is a real substitute for critical thinking, thorough analysis and following other software development best practices.

2. Overview of Tools

In order to manage a test project and handle composing and execution of tests, several tools and frameworks are used in the scope of this thesis. This chapter gives an overview of these tools. As Selenium is a primary part of this thesis it will described in more detail.

2.1 Introduction to JUnit

JUnit [21] is a framework to write unit tests on the Java platform. In case of JUnit framework the test will be written in the Java programming language and it will be used to declare and execute Selenium test methods. In addition, its methods will be used to verify the expected results. JUnit provides assertion methods for all primitive types and Objects and arrays [20]. For instance, a frequently used method in tests is `assertTrue`, which may be used to verify that a required element is present on the web page. JUnit library version 4.11 was used in this work. Tests and methods indicated in this thesis may not work with older versions of JUnit.

2.2 Introduction to Maven

Apache Maven [3] is a compiling and managing tool for Java programming language based projects. Maven configuration file for a project is written in XML format in a file named `pom.xml` located in the root folder of the project. A sample Maven configuration used in the scope of this thesis, to add JUnit framework and Selenium libraries is listed in Appendix 1.

2.3 Introduction to Selenium

Selenium [38] is set of software tools for web application test automation across different platforms, which is easy to use and flexible. One of Selenium's main features is executing tests on multiple browser platforms and each Selenium tool has its own approach to support test automation. In addition to browser based test script drafting Selenium also includes a specific test script language *Selenese*, which enables writing Selenium test

scripts in common programming languages and running those scripts against different browsers [33].

As stated in Selenium documentation it first came to life in 2004, WebDriver (see Section 2.3.1) project was started in 2006 and in 2008 the two projects merged into one providing the common set of features for all users [38]. There is an official user group, where people can get help, when having problems with Selenium or to find out if someone else has encountered the same issue. For people who are interested in contributing code to Selenium or just wish to help out, then all the information is available on the Selenium "Getting Involved" homepage [37]. In this work the Selenium library version 2.40.0 was used.

2.3.1 Overview of different Selenium tools

As already pointed out previously, Selenium is a set of software tools for web application test automation. The following will give a short overview of each of those tools separately. Commonly at least two of them are used together. All tests developed in the scope of this thesis have been written in Java programming language as repeatable automated tests using WebDriver tool.

Firstly, *Selenium IDE* is a Firefox plugin with easy to use interface for developing automated tests and is simply intended as a prototyping tool [38]. Selenium IDE [43] has support for exporting tests formatted in defined programming language for easier implementation in other tools. Selenium IDE can be launched in a separate window for script editing. Furthermore, Selenium IDE allows recording of the interaction with the web page and saves it into a test case. Recorded test case can then be edited, improved and run again. Selenium IDE is a convenient and user friendly tool for prototyping Selenium test cases and learning Selenium syntax.

Secondly, for a long time Selenium main project was *Selenium RC*, which consists of Selenium server and client libraries [39]. Selenium server is responsible for launching and killing browsers in addition to interpreting Selenium commands and running them on browser, acting as an *HTTP proxy*. Selenium RC server reports back with the results of

running these commands. A client library provides the interface between each programming language and the Selenium RC Server. The main focus of Selenium RC is to translate Selenese commands to corresponding programming language. To run Selenium RC its server and client libraries are needed.

Thirdly, *Selenium Grid* [42] allows one to run tests in different machines against different browsers in parallel. This can reduce the time it takes to complete a test suite. It is useful especially in agile development, where release cycles are short and developers need to verify their code after small changes. As of version 2.0 Selenium Grid was merged with the Selenium RC server. A grid consists of a single hub, and one or more nodes. When the hub receives a test to be executed along with information with which configuration the test should be run, it selects the corresponding available node. After that Selenium commands initiated by the test are sent to the node through the hub and executed within the browser against the application under test.

Lastly, *Selenium WebDriver* is the future direction of the project and the newest addition to the Selenium toolkit [38]. Selenium WebDriver was developed to better support dynamic web pages and its main goal is to supply a well-designed object-oriented API [41]. Selenium WebDriver drives the browser directly using each browser's native support for automation and for that it uses a separate driver library for each browser. WebDriver can be used in combination with Selenium server when there is a need to run tests on browsers in different machines. An easy way to set up Selenium WebDriver project is to use Maven.

When running tests WebDriver launches the browser, enabling the tester to follow the web application behavior based on commands executed by the Selenium test script. The Selenium WebDriver tool with Java programming language is used in this work. Java based Selenium library specification in Maven `pom.xml` configuration file is listed in Appendix 1.

2.3.2 An example Selenium test

Let us now see a simple example to understand how Selenium WebDriver tool can be used together with JUnit framework. For code development and JUnit test execution *Eclipse IDE* [12] will be used. In the scope of this thesis all the Selenium tests are run on Firefox browser, version 29.0.

To keep WebDriver management in one place and to ensure that the driver is initialized the same way for every test an abstract Java class was created. Having an abstract class enables creating simple and readable JUnit tests for developers who are not familiar with Selenium tests inner logic. In addition, having this class saves developers from writing a lot of the same methods in different test classes causing code repetition. The mentioned abstract class code is listed in Appendix 2.

The test class, used to demonstrate and explain Selenium code is listed in Appendix 3. To avoid restarting the browser it is launched once before executing test methods in the class. For this `@BeforeClass` annotation is added to method `createFFDriver`, which will launch the Firefox browser by initiating a new Firefox driver object and assigning its value to the global `Selenium.WebDriver` variable, which is used to interact with the browser. To ensure that the browser will be closed after running all test methods `@AfterClass` annotation is added to the method `cleanup`. This method will also use the previously mentioned global variable to close the browser instance with Selenium `quit` command.

The example test class contains only one test method `exampleTest`. Firstly, the URL, which in this test is the homepage of the Tallinn University of Technology, is opened in the browser by calling the Selenium command `get`. Then the title of the page is verified against the expected result. For this Selenium method `getTitle` is used, which returns the title of the current page, together with the JUnit framework `assertEquals`, which in case of disparity between the actual and expected value will stop the test execution and return a failure.

The test also checks that the search field is visible. Many functions interact with page elements to either get the value of their inner text or to click on them. For this WebDriver has method `findElement`, which returns a `Selenium.WebElement` object or throws an exception if a matching element is not found. This method takes a locator `By` as a parameter, based on what the object is searched. To ensure that the element is present on the page abstract class function `isElementPresent` uses Selenium `findElement` method and if it throws an exception then `false` is returned, otherwise the function will return `true`. The returned value is verified with JUnit method `assertTrue`.

As the next step the test types the string "cyber security" in the search field. Before that it is ensured that the field is empty by calling Selenium method `WebElement.clear` and then the text is typed to the search field using Selenium method `WebElement.sendKeys`. After that the test assures that there is a search button, which in case of Tallinn University of Technology homepage is a magnifier icon. For that method `isElementPresent` is used again. The search button is then clicked on by using the Selenium command `WebElement.click`.

After performing an action that requires a new page to be loaded a wait command is used to make sure that the content is present before executing new commands. To accomplish this `WebDriverWait` in combination with `ExpectedConditions` is used, which ensures a specific element is present on the page before continuing [44]. Finally the beginning of the title of the search page opened is verified against the expected value "*Otsing*" as already described before.

2.3.3 When to use Selenium

Although, it would be convenient to use a single tool to fulfill all testing needs, it is not usually possible. It is not reasonable or even feasible to manage all kinds of test automations with Selenium. Next we will take a look at what kinds of tests can be automated using Selenium and which tests should not be implemented with Selenium.

Selenium is an excellent tool for consistent testing of web applications user interface functionality in different browsers and on different machines. Moreover, in the course of functional testing with Selenium it is possible to confirm that each button or link on the user interface acts on a click as it is ought to. Selenium makes checking texts and field values displayed to the user straightforward. Selenium is not suitable for checking visual properties of user interface design as it lacks the ability to assure that the layout of graphical elements in different browsers will remain as required.

Selenium is also unsuitable for load testing. There are many reasons for that, but one of the main reasons is that load testing with Selenium, as it is not designed for load testing, is much slower than using some special tool for that. In addition, load testing with Selenium is not that accurate as the results will depend also on the speed of Selenium executing the test commands in the browser. Furthermore, Selenium should not be used for tests where Selenium's execution time can interfere with the results, like performance testing, because these test results are unreliable and not comparable.

2.4 Related work

While this thesis focuses on using Selenium for test automation, it is not the only tool available for web application test automation. Some of the other tools available for test automation are looked at in this chapter.

Ranorex [35] is a commercial test automation tool with a graphical user interface. It enables recording the tests and re-running them like Selenium. In addition, it has a straightforward report generation functionality. Another tool for GUI test automation is Squish [18]. Squish is a cross-platform/cross-technology test automation tool. Similar to Ranorex it is not free of charge and requires commercial license.

Besides the commercial tools, there are also more free alternatives available. For example, Watir [53] is an open-source family of Ruby libraries for automating web browsers and drives browsers the same way people do. While it enables testing web applications developed in Java, the tests need to be written in Ruby. Another tool similar to Watir is WatiN. WatiN [52] development was inspired from Watir and it can be used for test automation for web applications which are based on .Net languages. WatiN does not support writing end executing tests in Java.

There are more tools to choose from and some are platform specific meant specifically for Android platform, for example, Android GUITAR [48] or work only on Windows operating systems, for example, Axe Test [31]. Then there are several commercial tools, for example, QA Wizard [36] that work on different platforms and support a wide variety of browsers, but require a commercial license. Selenium stands out because it is an open source project that can be used free of charge and supports different platforms and different browsers with different tools to choose from.

Studying and experimentally confirming the support of authentication methods provided by all these tools would require a non-trivial amount of work probably sufficient for another thesis. Therefore, as the focus of this thesis is on Selenium, these features will not be discussed here.

This concludes the part of the thesis where the relevant tools are introduced. In the following an overview of several authentication schemes used in web applications is given while also pointing out their advantages and disadvantages, which are important to consider when deciding which authentication scheme to implement. In addition, some security aspects are discussed that need to be taken into account, when using the authentication methods.

3 Introduction to authentication schemes

Authentication effectiveness depends on the effectiveness of the authentication protocol and its fundamental proof of the authority [19]. Hence it is very important to choose and implement secure and reliable authentication scheme appropriate for the application. In the following we will take a look at HTTP Basic, login form based authentication, user certificate, Estonian ID-card and Estonian Mobile-ID authentication schemes.

3.1 HTTP Basic

In the Figure 1 basic overview of how this authentication method works is shown.

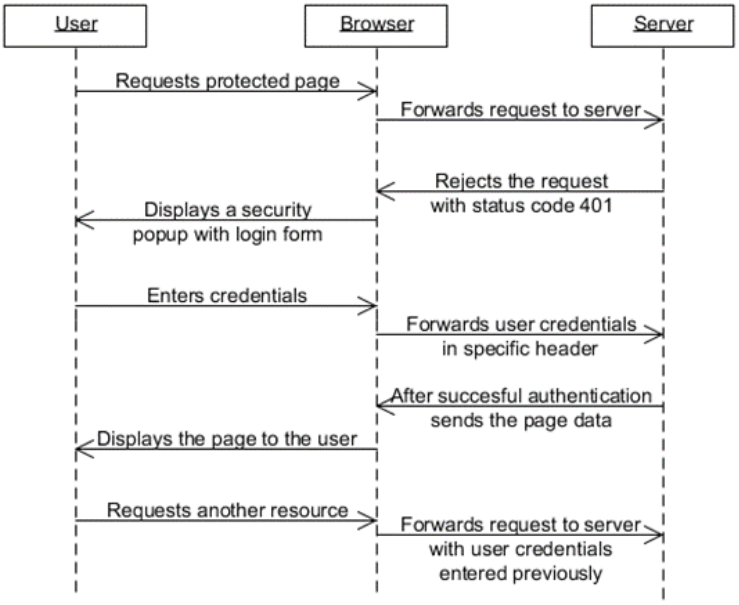

Figure 1. HTTP Basic authentication sequence diagram

To summarize an HTTP Gallery article [46], HTTP Basic authentication is a simple way to regulate access to web pages. If the server receives an anonymous request to a protected page, it forces an HTTP Basic authentication by rejecting the request with status code 401 and setting the `WWW-Authenticate` header values. `WWW-Authenticate` header will contain the word Basic, which will determine the authentication mechanism that the HTTP client must use to access the corresponding page. In addition a realm string value will be set, which can contain any value to identify the secure area.

Many browsers will display a login screen based on that reject and send the user provided username and password in base64 encoding. The encoded string will represent the user's credentials in a form of `username:password`. The string will then be sent to the page in an `Authorization request` header.

A weakness in HTTP Basic authentication is the fact that it sends the credentials over the network in clear text, unless the connection is encrypted using other means such as TLS or VPN. This makes it possible for third parties to obtain user credentials [16]. Therefore, HTTP Basic authentication should never be used without secure or encrypted communication channel.

3.2 Login form based authentication

In the Figure 2 basic overview of how this authentication method works is shown.

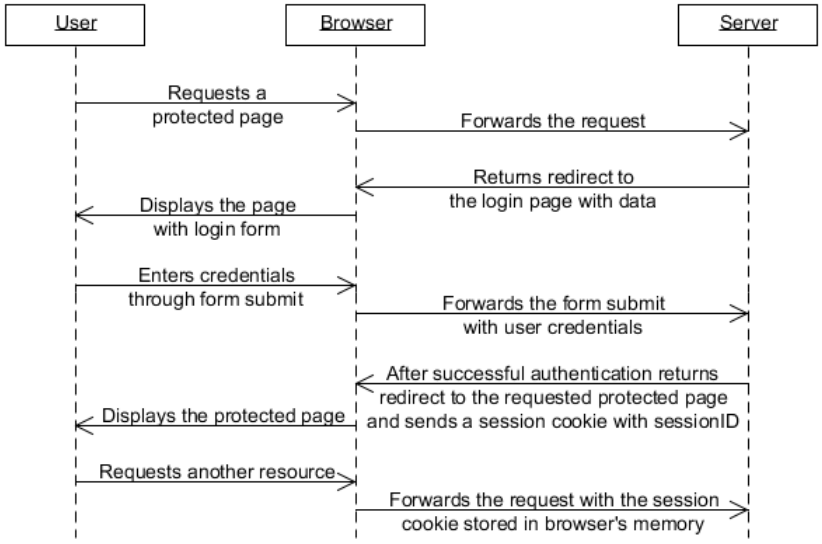

Figure 2. Login form authentication sequence diagram

HTTP is a stateless protocol, meaning that user data is not persisted from one page to another on the website and one way to manage user information is through the use of cookies [27]. For session management cookies, which will be kept in the browsers memory until the browser is closed or the session is ended, can be used. A separate session with unique session cookie will be created for each user. To determine which rights the user should have, the user needs to be identified. For this login form based authentication can be used.

In the case of login form based authentication, when the user requests some protected page he is first directed to a login page usually containing a simple form with two fields and a submit button. These fields are for user ID and password. After the user enters these values the system compares them with values previously saved on the server. If the combination of these two matches the values known to the server the user is successfully authenticated and authorized.

After a successful authentication the server will generate a new session to avoid session fixation attacks [23] and send the session cookie to the browser. The browser will present the cookie to the server with every following request enabling the user to interact with

the application during the session without having to authenticate again before every request [47]. Usually a specific and unique ID is assigned to one concrete session and this is kept persistent throughout the entire session in the session cookie. On the server side all required information about this session is kept, for example the logged in user's username and language preference. As a first thing, when a browser sends the cookie with the request to the server, it is first determined if the user has already been authenticated and that the session with the ID is still active [2].

When using this approach for authentication some security aspects should be taken into consideration. Firstly, session cookies are prone to several attacks like brute force [14] and XSS attacks [32]. When a third party is able to capture an active session cookie, the attacker is able to act on behalf of the user and access all information available to the user under attack. To prevent cookies from being observed by third parties due to the transmission of the cookie in clear text it is mandatory to use an encrypted HTTPS connection for the entire web session [45].

An advantage of using login form based authentication is that cookies are easy to create and manage, making implementing this approach straightforward [34]. In addition implementing a separate page with a form for the user to present his credentials and comparing them with previously stored values is also rather easy to achieve. For users this authentication measure is simple and understandable. This is a frequently used authentication scheme in web applications.

3.3 User certificate based authentication

In the Figure 3 basic overview of how this authentication method works is shown.

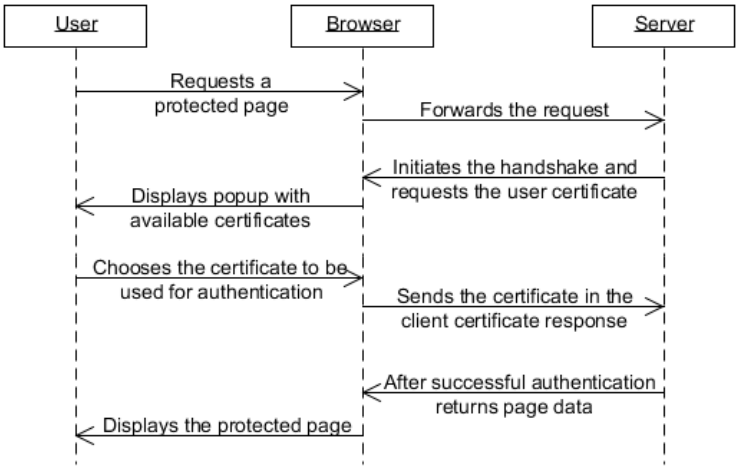

Figure 3. User certificate authentication sequence diagram

The actual handshake performed is more complicated and shown in Figure 4 [17].

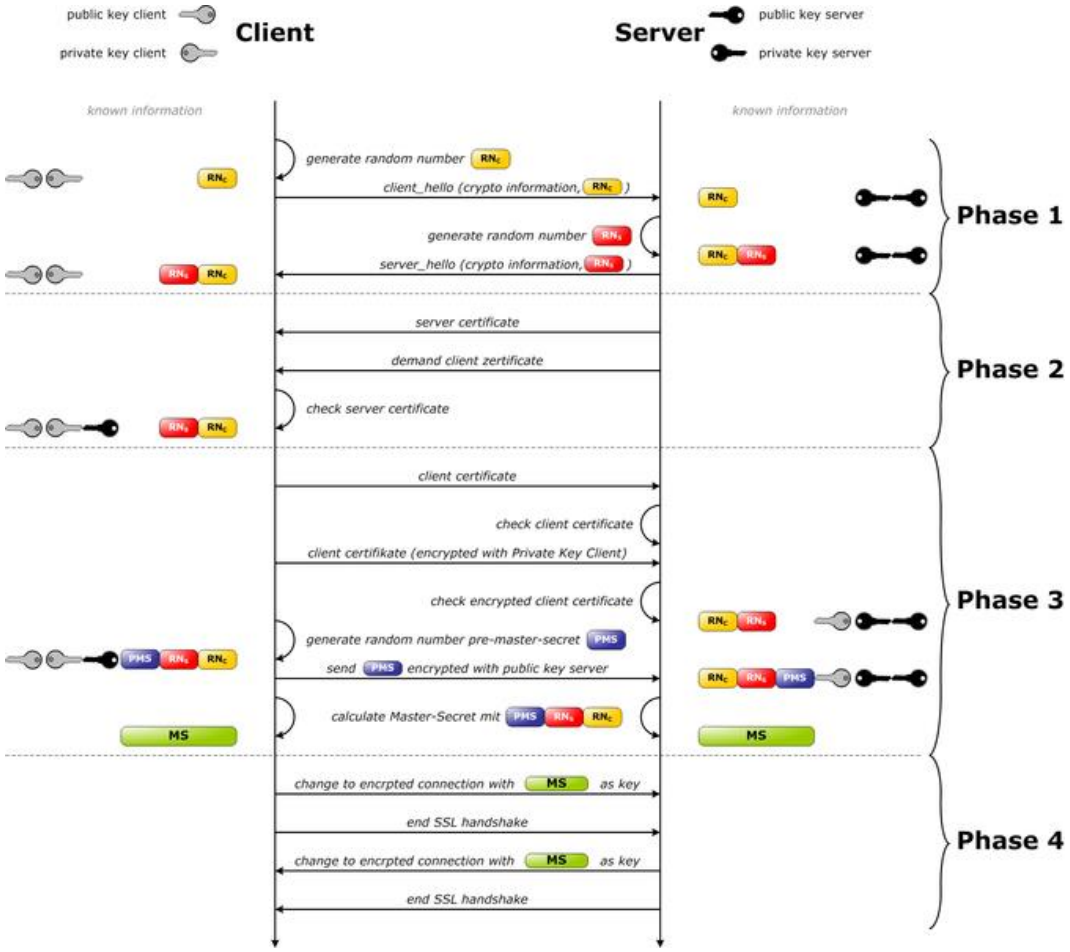

Figure 4. SSL handshake with two way authentication with certificates

User certificate based authentication as the name indicates is authenticating the user to the server and establishing the authenticity of the user via public key cryptography and user certificates. When accessing a protected page over HTTPS with configured user certificate based authentication TLS Handshake protocol is used [11] and during the handshake the client and server exchange their public key certificates to verify their identity.

The certificates can be self-signed or signed by some Certificate Authority (CA). To make development of web applications with user certificate based authentication easier HTML5 introduces a new markup element `keygen` [9], which represents a control for generating a public-private key pair and for submitting the public key from that key pair to be signed by the server. This approach helps to save time and money as there is no need to get the certificate signed by some authorized CA.

The advantage of user certificate based authentication is that user does not need to remember a password for the site. Instead a certificate is usually held on the computer's hard drive and imported into the browsers memory. When the site requests user authentication the browser accesses the certificate and manages the authentication on behalf of the user. Usually the browser asks for the user to specify the correct certificate that should be used with the corresponding website if not configured otherwise. Furthermore, unlike with login form based authentication, where the server side also keeps the user credentials, with certificate based authentication the server does not store any secret information as the secret key is kept on the client side.

On the other hand, certificate based user authentication has its disadvantages and may not be the first choice for some websites. When the web application should be accessible to user from different machines it is not a good solution. This is because it will require setting up the certificate on all machines or forces users to use the same computer each time they access the website.

3.4 Estonian ID-card

In the Figure 5 basic overview of how this authentication method works is shown.

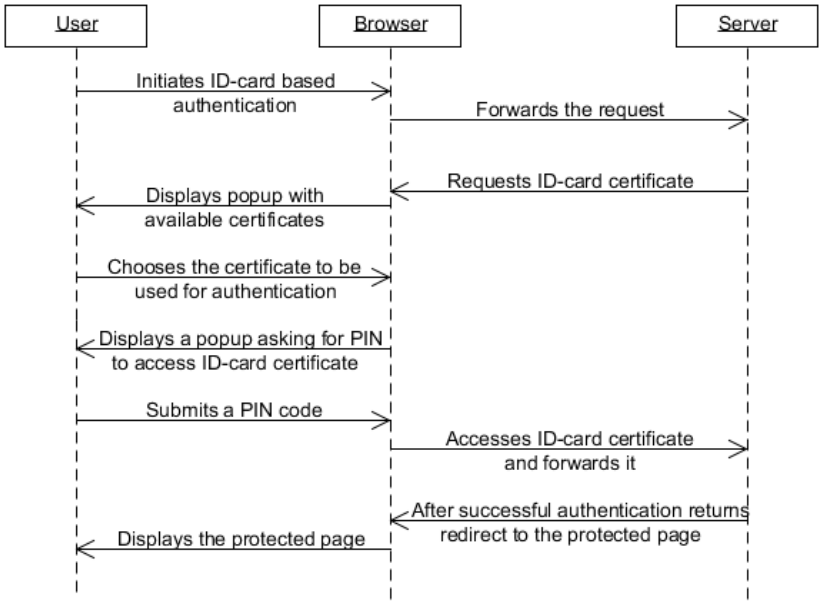

Figure 5. Estonian ID-card authentication sequence diagram

Estonian ID-card is a legal personal identity document, which can be used in the real world as well as in the digital world. For example using an ID-card is a convenient and safe way to access internet banks and using other sites holding highly confidential information. Furthermore, ID-card is safer than simple user certificate because it uses a two factor user identification system. In addition to owning a physical card containing the certificates and corresponding private keys, the user has to know a four digit PIN code. Without the PIN there is no way to use the certificates on the ID-card for authentication.

The authentication process works as follows [30]. Estonian ID-card chip contains two keys, public and private key along with the user certificate. The public key and user certificate are on the public part of the chip, where it can be accessed for example through Public Key Infrastructure. The private key is stored on the secure side of the chip, where it can only be accessed with PIN codes. These two keys are connected to each other mathematically, but there is no way to derive private key from the public key. While, sent messages encrypted with the public key can only be read by the receiver who has the private key, messages encrypted with the private key can be validated with the public key.

During the authentication with ID-card the server sends the session key encrypted with the ID-card public key. This session key is in return decrypted with the private key. After that a secure connection with the server is enabled and encrypted packets will be sent over the physical network. If the user certificate is invalid (for example, expired or revoked) or the user inserts a wrong PIN code the session key exchange fails and the connection with the server will also fail. In order to enable the use of Estonian ID-card based authentication the user has to install special software to his computer and attach the ID-card reader with physical card to the computer.

In order to develop Web applications with Estonian ID-card based authentication support `DigiDoc` libraries can be used, which are available on the ID homepage [4]. At the moment of writing this thesis there were five libraries available supporting programming languages like Java and C. In addition to libraries, for electronic signing support, the *DigiDocService SOAP-based* web service is available.

On the one hand, based on the ID homepage this authentication scheme has many advantages [6]. It is a better and more secure measure than login form and user certificate based authentication. Moreover it lowers the risk of somebody presenting the false identity to the website as the user has to have personal ID-card and has to know the PIN code assigned to it. Even if the physical card is stolen from the user it cannot be used in fraud without the PIN. From the user's side of view it is convenient as there is no need to remember many different usernames and passwords as one card and one PIN can be used for authentication on many websites.

On the other hand, in order to use ID-card for authentication the user has to have a special ID-card reader attached to the computer, which requires specific device drivers to be installed on the computer beforehand as covered previously. Therefore it may be inconvenient from the user's point of view compared to login form based authentication, for example. Although many people carry their ID-card with them daily it is not convenient when going abroad as there is also a need to carry the ID-card reader.

3.5 Estonian Mobile-ID

In the Figure 6 basic overview of how this authentication method works is shown.

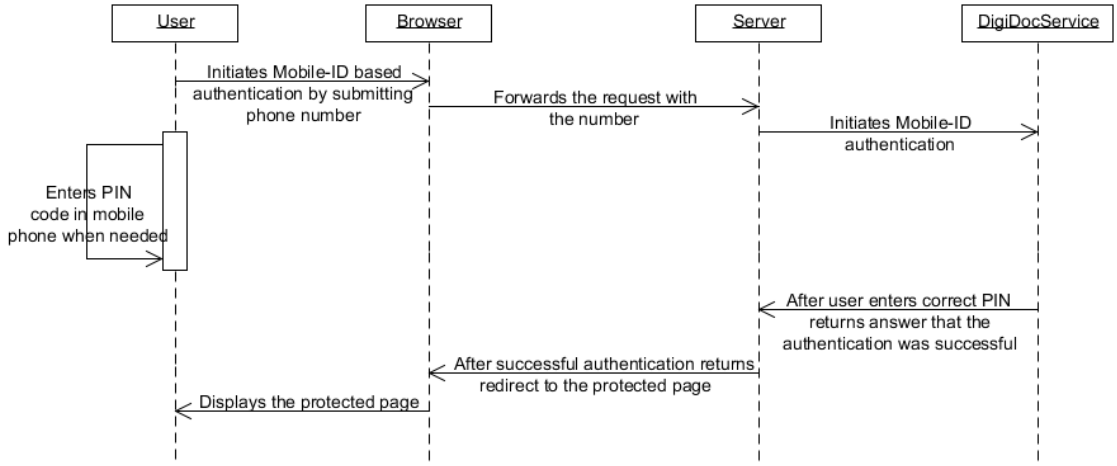

Figure 6. Estonian Mobile-ID authentication sequence diagram

Mobile-ID was first introduced in May 2007 by mobile operator EMT and it could only be used in private sector applications in Estonia. In 2011 mobile-ID was recognized as a national document for user identification [25]. Mobile-ID is a similar user identification solution as ID-card with two certificates for identification and other one for digital signing. In case of Mobile-ID these certificates and public-private key pairs are placed on the mobile SIM-card [7]. One person will be associated with one set of certificates and a SIM-card can only contain one set of certificates.

Data exchange between the mobile phone and the corresponding web service environment is done over encrypted connection [7]. According to the DigiDocService specification in order to raise security for Mobile-ID authentication web applications have to clearly display a verification code to the user and ask the user to check the code before entering the PIN code in the phone [5]. If the verification code on the website and in the phone do not match, then the user is obligated to terminate the authentication process and avoid entering the PIN code. Although, the authentication process with Mobile-ID is similar to ID-card authentication process, for Mobile-ID authentication support SOAP-based RPC-encoded DigiDocService web service has to be used.

On the one hand, Mobile-ID has its advantages over ID-card based authentication. Mobile-ID is easy to use and more convenient than ID-card, because it works everywhere with mobile phone coverage. In addition there is no longer the need to worry about installing necessary software on your computer or purchasing card readers as with Mobile-ID the software will be in your mobile phone. Furthermore, all modern mobile phones are suitable for Mobile-ID and its security level is considered equal to ID-card security level, so it can be trusted for authentication [7].

On the other hand, one disadvantage or security risk for Mobile-ID is the vulnerability of the mobile phone. In case of keystroke logging threat on the mobile phone the PIN value for Mobile-ID authentication can easily be captured and forwarded on to a remote location [10]. While this is a major threat to Mobile-ID authentication, the PIN code cannot be used without owning the corresponding SIM-card.

4 Automating authentication schemes tests with Selenium

In this chapter the focus will be on automating previously introduced authentication schemes with Selenium. An overview of how to implement these tests with code examples are given. All code examples represented in this chapter are in Java programming language written as JUnit tests using Selenium WebDriver tool for browser management and executing commands in the browser.

4.1 HTTP Basic

4.1.1 Overview of the application under test

As already covered in Section 3.1 HTTP Basic is a simple way to regulate access to web pages. For the application under test (AUT), against which the Selenium test will run, a simple example web page with HTTP Basic Authentication is used. For authentication implementation Spring Security [51] framework is used. Moreover, the version of Spring Security used in the AUT is 3.0.5.RELEASE. Spring Security configuration is located in the XML file named `spring-security.xml` and to configure HTTP Basic authentication the lines shown in Figure 7 are added to the file.

```
<http>
      <intercept-url pattern="/*" access="ROLE_USER" />
      <http-basic />
</http>
```

Figure 7. HTTP Basic Spring Security configuration

This is the simplest and minimal configuration needed to restrict access on the request URL that matches the pattern specified in the configuration. Requests matching the given pattern must fulfill access requirements set in the access attribute [1]. Based on this configuration every request made to specified path requires a user with given role `ROLE_USER` to be authenticated using HTTP Basic authentication.

Firstly, the AUT will run in the same computer as the tests. The application and the Selenium tests are run using Eclipse IDE. The project is run under Eclipse Java EE perspective with pre-configured Apache Tomcat application server [50] instance. Tomcat

version 7.0.53 was used to execute applications under test. The application server is started before executing any tests.

As described in the Section 3.1 many browsers display a login form to insert username and password based on the `WWW-Authenticate` header sent by the server for HTTP Basic authentication. The AUT site is accessible from the URL `http://localhost:8080/HTTPBasicTest/`. When opening the AUT web page in the Firefox browser we see the form and fill it with test username and password as shown on the Figure 8.

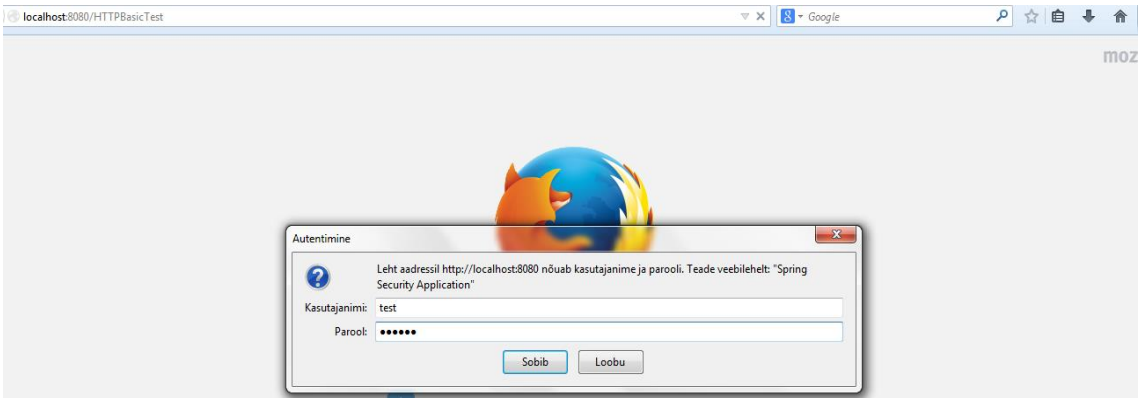

Figure 8. HTTP Basic authentication form

After inserting the test users username and password values the corresponding role is assigned to the user and he is redirected to the page requested. Based on AUT Spring Security configuration for user with ID "test" this role is `ROLE_USER`. After successfully being authenticated a simple web page is displayed with title "HTTP Basic Test Page".

4.1.2 Implementation and execution of the Selenium test

The Selenium test supporting HTTP Basic authentication has a similar structure to the example described in Subsection 2.3.2 – the test uses the same abstract base class for common functionality. The Firefox browser instance is also started and stopped identical to the example test. The source code of this test is listed in in the Appendix 4.

Firstly, the URL is specified for the test. As Selenium cannot see the security popups displayed by the browser the test cannot enter the username and password through the displayed form. For testing with Selenium there is an option to send the credentials in the

URL in a format `http(s)://username:password@url`. In the AUT the username for a valid user is “test” and the password is “123456”. Keep in mind that this is a test project for this thesis and the password should never be that trivial. While taking all this into consideration the URL used for this Selenium test is shown on the Figure 9.

```
String url = "http://test:123456@localhost:8080/HTTPBasicTest/";
```

Figure 9. URL set for HTTP Basic automated test.

Secondly, the test code contains method `createFFDriver` annotated with `@BeforeClass` annotation, which will setup the Firefox. Then starting JUnit test execution this method is executed first. After this method has run an empty Firefox window will be displayed in the browser started by the Selenium driver.

Thirdly, the test starts the same way example test started by opening the URL specified previously. As the user credentials are sent in the URL to the server, no authentication popup is displayed. The user requesting specific page is authenticated based on the credentials described in the URL. Again the `waitUntilLinkIsVisible` command is used to assure that the page has time to load before continuing with the test. Moreover, as it is known that the page has to display a logout link the line demonstrated in Figure 10 is added to the test execution.

```
waitUntilLinkIsVisible("Logout");
```

Figure 10. HTTP Basic logout link wait command

The user is authenticated and given access to the site based on the URL. This means there is no need to add additional steps to the test to enter username and password. For Selenium test the page displayed is already the web page of the application. Page contains the username of the authenticated user and a logout link as shown in Figure 11.

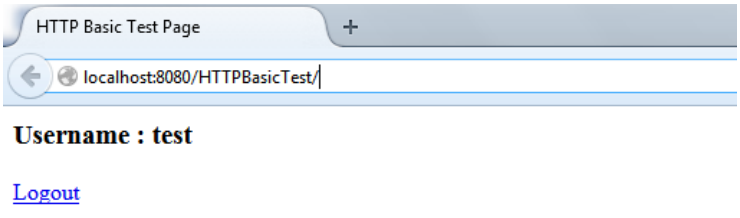

Figure 11. HTTP Basic AUT web page

In addition, to verify that the user is indeed successfully authenticated and a correct page is displayed in the browser window some assertions are added. As a first thing it is ensured that the page title has the value expected. After that it is verified that the text displayed between `<h3>` HTML tags is also the one expected and contains the right username value. This is achieved by adding the two lines shown in Figure 12 at the end of the HTTP Basic authentication scheme Selenium test method `httpBasicTest`.

```
assertEquals("HTTP Basic Test Page", getPageTitle());
assertEquals("Username : test", getHeadingText("h3"));
```

Figure 12. HTTP Basic AUT page verifications after successful login

Lastly, to clean up after the tests and close the opened browser the method `cleanup` with annotation `@AfterClass` is added. Identically to the example test it calls the abstract class method `quitBrowser`. After this method is executed the browser is closed and the JUnit test execution exits with successful result. In Eclipse IDE "JUnit" tab the output demonstrated in Figure 13 is displayed.

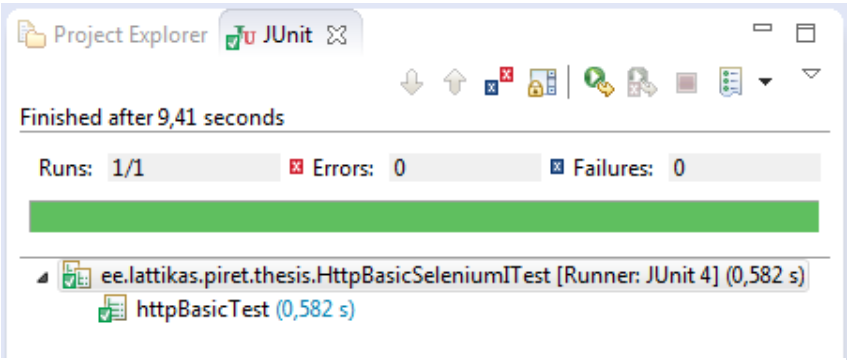

Figure 13. Eclipse IDE JUnit test execution results

4.1.3 Summary of the authentication scheme test

On the one hand automating HTTP Basic authentication scheme tests with Selenium is rather easy and does not require much code writing. One thing that differs testing application with HTTP Basic authentication and regular application without authentication is the format of the URL sent to the protected page. In addition this makes testing the authentication with different user credentials also straightforward.

On the other hand this test raises many issues and security concerns. Firstly, multiple login attempts and login failures cannot be tested with this approach. If the first attempt fails, meaning the URL contains false credentials, then the browsers security popup will be displayed again for another login attempt. And as already covered before the main flaw in test automation for the web applications with HTTP Basic authentication with Selenium is that Selenium WebDriver does not see the browsers security popups and cannot interact with them.

Moreover, as mentioned in the Section 3.1 HTTP Basic authentication scheme should not be used to protect valuable and confidential pages because of its security issues. A major security issue with testing it with Selenium is that the password and username are sent as clear text. This makes it vulnerable to attacks, as it is relatively easy for third parties to read the user credentials. Of course this test project for demonstrating HTTP Basic authentication could be improved by setting up an encrypted network connection and restricting the number of login attempts.

Furthermore, from the testing side this Selenium test is not as realistic as it should be. Inserting the user credentials for authentication is bypassed by adding them to URL. This makes testing HTTP Basic authentication based browser security popup displaying impossible. In addition there is no way to verify the value of the realm displayed to the user. Making it hard to test different browser reactions to the `WWW-Authenticate` header and find possible errors in the realm value through automated testing.

4.1.4 Conclusion

To sum up, automating HTTP Basic authentication scheme testing with Selenium is possible and quite easy to implement. This makes writing automated Selenium tests for sites protected with HTTP Basic authentication simple. It would be helpful if the Selenium WebDriver tool supported interaction with browser popup windows and this is a limitation in Selenium that should be addressed in the future.

4.2 Login form based authentication

4.2.1 Overview of the application under test

Similar to the HTTP Basic authentication the application under test (AUT), against which the Selenium test will run, is a simple example web page with login form based authentication. In case of this application also Spring Security was used to implement authentication on the pages with restricted access. Configuration added for login form based authentication in the AUT is shown in Figure 14.

```
<http auto-config="true">
      <intercept-url pattern="/admin**" access="ROLE_USER" />

      <form-login
            login-page="/login"
            default-target-url="/welcome"
            authentication-failure-url="/login?error"
            username-parameter="username"
            password-parameter="password" />
</http>
```

Figure 14. Login form based authentication Spring Security configuration

From this configuration it can be seen that the access is restricted on the admin site and only users with role `ROLE_USER` are allowed access to that page. Another thing to notice is that in this AUT a page has been specified in case the authentication fails. In that case the user is redirected back to the login page with a corresponding error message displayed. An identical runtime environment to the previous example is used here.

After the tomcat is started the AUT is accessible from the URL `http://localhost:8080/LoginFormTest/`, which is not a protected page and opens a welcome page. The protected page is accessible from the URL `http://localhost:8080/LoginFormTest/admin`, which in the case when there is no valid session found redirects the user to the login page based on the Spring Security configuration.

After inserting correct user credentials to the form and submitting it, the user is authenticated by assigning him the role `ROLE_USER`. The same user credentials are used for login form based authentication as in Section 4.1. After the user has been successfully

identified and authenticated he is redirected to a simple admin page containing a message with text “This is protected page!” and a logout link. In the following the Selenium test for this authentication scheme will be presented.

4.2.2 Implementation and execution of the Selenium test

Let us now focus on the implementation details of the test using form based authentication. The general outline of the implementation of the test is similar to the example discussed in Subsection 4.1.2. The full source code of the test class is listed in Appendix 5.

As described above, the page that is protected in the AUT is the admin site. As the test is intended to test login form based authentication specifically, then the URL will be set for the admin site. Taking this into consideration the URL used for this test is demonstrated in Figure 15.

```
String url = "http://localhost:8080/LoginFormTest/admin";
```

Figure 15. URL used for login form based authentication test execution

Firstly, the test method for login form based authentication `loginFormTest` starts with opening the URL pointed out before. Based on the Spring Security configuration for the AUT the URL will redirect the user to the login page. Knowing that the login page contains a heading with text “Login Form” a call for the abstract class function `waitUntilHeadingIsVisible` is added to ensure that the page is loaded before continuing with test execution. This method is realized with Selenium methods described in Selenium example test (see Subsection 2.3.2), the code for it is shown in Figure 16.

```
protected void waitUntilHeadingIsVisible(final String heading) {
      (new WebDriverWait(driver, 30))
            .until(ExpectedConditions
                  .textToBePresentInElementLocated(By
                        .cssSelector("h1"), heading));
}
```

Figure 16. Method to wait for page to load until specific heading is visible

Secondly, the login form contains a username and password field and a submit button to forward the credentials to the server for authentication. Before filling the fields with values there should be a check that the page loaded correctly and required fields are visible. To accomplish this JUnit method `assertTrue` is used to verify the answers from calls to abstract class methods `isUsernameFieldVisible` and `isPasswordFieldVisible`. These methods call the method which uses Selenium `WebDriver.findElement` function to find the elements by giving in the locator `By.name`.

After it is verified that the required fields are present user credentials are typed to the fields. Firstly the credentials of the non-existing user are entered, to demonstrate the page displayed then the wrong username and password are entered. After adding the bogus values to the fields the page displayed is brought out in Figure 17.

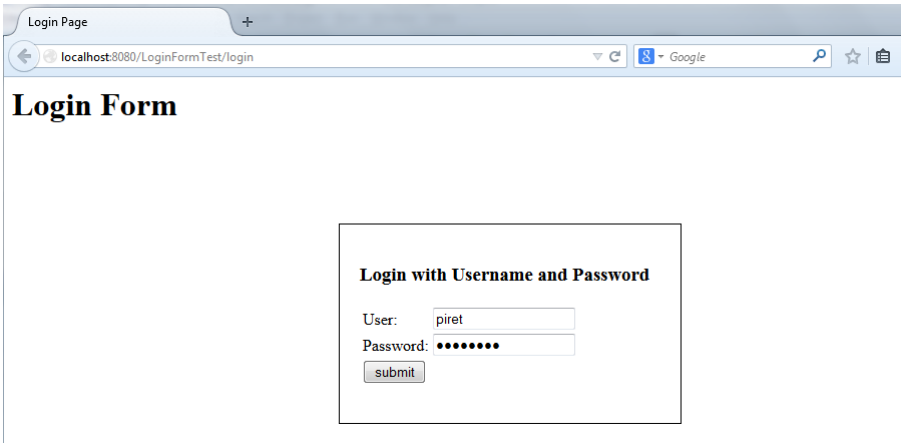

Figure 17. Login Form AUT login page

The submit button visible on the picture above is clicked. After that the expected action is that the user is redirected back to the same page and a corresponding error message with text “Invalid username and password!” is shown. For that we add lines to the test to assert the expected value of the page title and verify the value of the error message displayed between tags with assigned class containing name “error”. The error displayed with the form is pointed out in the Figure 18.

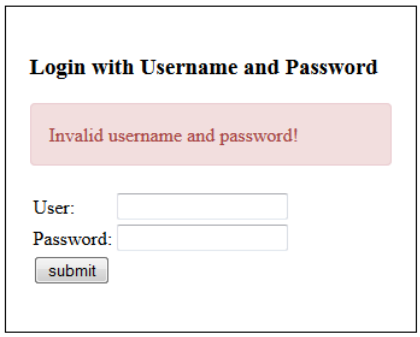

Figure 18. Login form AUT login page error

Thirdly, after verifying that the correct data is displayed to the user, after inserting the wrong credentials, it is verified that after fixing the username and password values the user is still able to successfully access the restricted area. For that the test next fills in the displayed form with valid credentials which in this AUT are "test" for username and "123456" for password.

Some websites containing login forms do not clear the wrong values typed when displaying an error message. That is why it is important to remove the old values before entering correct values to the login form fields. For this Selenium method `WebElement.clear` is used in the method to enter a new username value as shown in Figure 19.

```
protected void typeValueInUsernameField(String username) {
      WebElement field = driver.findElement(By.name("username"));
      field.clear();
      field.sendKeys(username);
}
```

Figure 19. Method to type specified value to the username field.

After that submit button is clicked again. The expected result is that the user is authenticated and given access to the admin page. Before asserting that the displayed page is the correct one, the test waits until the logout link is visible. For this a similar method `waitUntilLinkIsVisible` is used as already mentioned before. The page displayed to the user is shown in Figure 20.

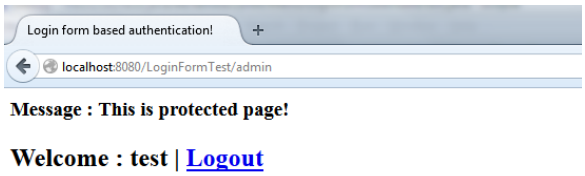

Figure 20. Login form AUT admin page

Lastly, the test ensures that the displayed page is the one expected. In addition to the logout link the page should contain the heading "Message : This is protected page!", which is searched form between `<h3>` HTML tags. After asserting the title of the page displayed and the heading mentioned before the JUnit test method exits with success and the Firefox browser used for the test is closed.

4.2.3 Summary of the authentication scheme test

On the one hand testing login form based authentication is easy and does not require much code writing. All that is needed is interaction with one form, by inserting two values to specific fields and clicking one button. In addition the test does not require special setup of the browser, the basic Firefox browser launched by the driver is enough.

Moreover, Selenium enables testing of the login form in a way it would be used by a user accessing the page. There is no need to somehow mock the login or like in case of HTTP Basic send the credentials with the URL. This enables testing the login form functionality and behavior in different browsers and get actual feedback about it. This is especially important then login is done using custom JavaScript or developers wish to test Ajax based requests and responses in different environments.

Furthermore, the advantage of using Selenium to automate login form based authentication testing is the ability to test error messages displayed as demonstrated. In addition there is also the possibility to test the behavior of the application in case the user exceeds the allowed number of login attempts. In this case, if the application locks the user out for some minutes, the Selenium test can be programmed to wait for the required time and verify that login is possible again after that time and not a minute before.

On the other hand this Selenium test should not be run with the actual user credentials, because the username and the password are written into the test code. In case some unauthorized person is able to access the test code it is rather easy to gain access with the data included in the tests. These tests should always be run against some test or development environment with values specific for testing.

Lastly, some companies may wish to keep actual data also in test environment to ensure that, based on the test results in there, the reliability of the application in live environment could be assessed. In that case, to avoid hard-coding passwords in test code, an external method (e.g., loading from a configuration file or getting values from the execution environment values) for storing and obtaining the usernames and passwords at test runtime should be used.

4.2.4 Conclusion

To sum up, automating the login form based authentication with Selenium is rather simple to accomplish. In addition it does not require additional setup for the browser or changes in the URL used to access the website. This makes test results gained from these automated tests comparable to application behavior in real situations and helps to avoid mistakes in authentication scheme implementation early on.

Although, no shortcomings in Selenium framework for login form based authentication testing were identified, additional measures should be used to secure the test environment. Before setting up the environment, where to run Selenium automated tests, testers should make sure that the confidential and valuable information is protected from unauthorized persons. Overall Selenium WebDriver is a great tool to use for test automation for web applications with login form based authentication.

4.3 User certificate based authentication

4.3.1 Overview of the application under test

For user certificate based authentication application under test (AUT) a company's Jenkins page was chosen. Jenkins is a continuous integration solution, which can be used to manage projects builds, releases, testing and monitoring the run of these jobs [22]. The AUT has restricted access on the page so that only company employees with certificates have authorization to see the website.

Before the protected website can be accessed with Firefox the certificate has to be imported to the browser. Then trying to access the page the server initiates the authentication by requesting the user certificate. The AUT is accessible from the URL `https://office.zerotech.ee/`. When the server requests for user certificate the browser opens a security popup displaying information about available certificates. The popup displayed in Firefox is presented in Figure 21.

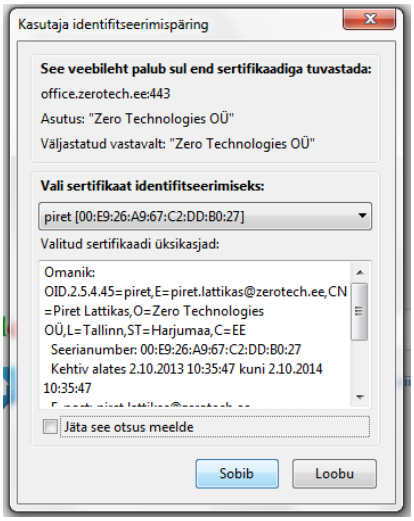

Figure 21. User certificate browser security popup

After selecting the correct certificate and being successfully authenticated in the server the user is directed to the protected page. The page in case of this AUT page is the Jenkins dashboard. In the following additional steps needed to setup Firefox driver for automating tests with Selenium for user certificate based authentication are demonstrated.

4.3.2 Preparations needed for the test implementation

As already mentioned in Section 4.1 Selenium driver cannot see and interact with browsers security popups. Because of this before testing the website with user certificate based authentication there is a need to import and enable a specific certificate for the AUT. Selenium has support to specify specific driver profiles and following will show exact steps needed to specify a Firefox profile for AUT.

Firstly, the custom Firefox profile needs to be saved to a folder where the test code can access it. For this a new folder with name `FFProfile` is created under the test projects `src/test/` directory. In case of testing different certificates, different profiles have to be created under different folders for each user certificate.

Secondly, a Firefox profile has to be created. Mozilla support page has a description on how to open the Firefox Profile Manager and use Create Profile Manager to enter a new profile [29]. For this test custom profile named `user_certificate` was created and the folder specified for this profile is the one created previously. Now to add user certificate to the profile the browser is launched choosing the custom profile.

Thirdly, Firefox options pane is opened and tab "Advanced" is chosen. From there tab "Certificates" is opened. Now to make sure the browser would skip asking the certificate while running the Selenium test the option "Select one automatically" as demonstrated in Figure 22 is set.

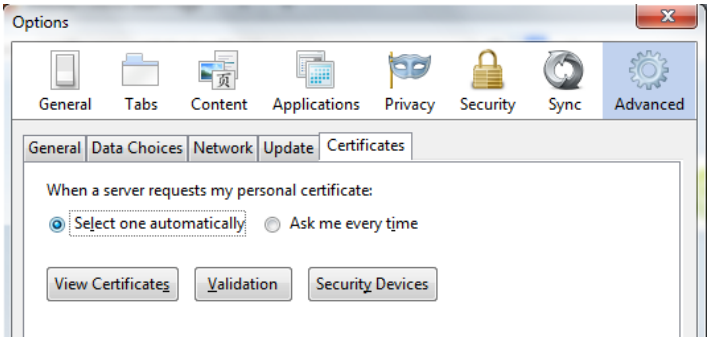

Figure 22. User certificate browser security popup

The certificate has to be imported under that profile. For that on the same pane the “View certificates” button is clicked and tab “Your certificates” is chosen. For a new profile the list of certificates should be empty. The test certificate is imported by clicking on the “Import” button and locating the `.p12` certificate file and clicking “ok” button. This is a manual process.

After that the certificate popup to set the master password is displayed. For automated testing purposes the password needs to be left empty. Otherwise before sending the user certificate to the server the browser will ask for a password through browser security popup. So without making any changes “ok” is clicked. The browser prompts a message that this is not secure, but for testing purposes it should be ignored. One last thing that needs to be done before the certificate is successfully added to the list is to enter the password assigned to the certificate so that Firefox could access it. Now the certificate has successfully been imported and can be seen in the list of “Your certificates” as show on the Figure 23.

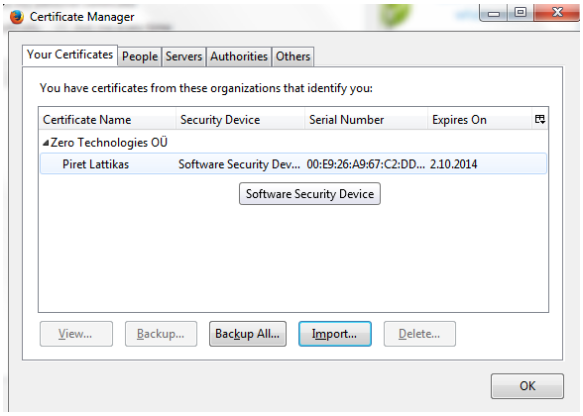

Figure 23. User certificate imported to Firefox profile

Lastly, to ensure that Firefox browser is able to bypass the untrusted connection page it has to be trusted under the profile. To do that the AUT restricted page should be opened while still accessing Firefox browser with the custom profile. So URL `https://office.zerotech.ee/` is typed to the address field and as expected the browser displays the “This Connection is Untrusted” page. From there the “I Understand the Risks” should be chosen and button “Add Exception…” is clicked to confirm the security exception. Now this page is trusted and automatically authenticated by the browser. Following will demonstrate how to implement the Selenium test based on the custom profile.

4.3.3 Implementation and execution of the Selenium test

To begin with test automation for web application with this authentication scheme with Selenium the first thing as with other tests is to specify the URL to be used. The source code of the test class used for this test is listed in the Appendix 6. As can be seen from the test class code for this test a different method `setUpFFDriverWithProfile` is used to setup and launch the Firefox browser. The major difference with driver setup is that for this test custom profile is used to bypass certificate related browser security popups. The Firefox driver is initiated with the lines of code shown in Figure 24.

```
@BeforeClass
public static void createFFDriver() {
      // Launch the Firefox browser using Firefox driver.
      FirefoxProfile profile = new FirefoxProfile(new
           File("src/test/FFProfile"));
      setUpFFDriverWithProfile(profile);
}
```

Figure 24. Method to initiate Firefox driver with user certificate specific profile

Moreover from this code it can be seen that the Firefox profile is created based on the setup created previously under the folder `src/test/FFProfile`. After initiating a new profile it is passed as a parameter while creating a new instance of the Firefox driver. When the driver is initiated a new Firefox browser instance has been launched showing a new blank tab.

Firstly, as with previous test the URL specified before is opened. In case of this test, as we set the Firefox browser to automatically select the certificate for the websites, no security popup is displayed to select the certificate. Instead the browser selects the certificate and sends it to the server for authentication. So the user is authenticated and authorized on the background. The connection to the corresponding page is already trusted and security confirmation of that has been saved into the custom profile. Moreover, based on this the first page displayed should already be the expected Jenkins dashboard page, which is partly demonstrated in the Figure 25.

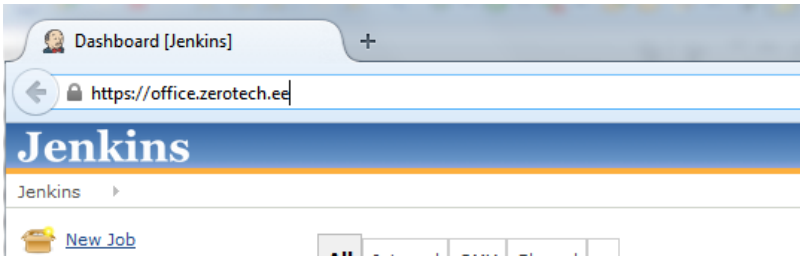

Figure 25. User certificate AUT page

Furthermore, as can be verified from the Figure 25 the test page contains a link with text "Jenkins". So before asserting if the website is the one expected the test waits first until this link becomes visible on the page. After making sure the page has loaded it is verified that the page title equals with the expected one and link to add new jobs to Jenkins is available. This is realized with adding the lines displayed in Figure 26 at the end of the test.

```
assertEquals("Dashboard [Jenkins]", getPageTitle());
assertTrue(isLinkWithTextVisible("New Job"));
```

Figure 26. User certificate AUT page verifications after successful login

Lastly, to verify the link is visible with the text "New Job" the abstract class method `isElementPresent` is used with giving in the `By` locator as a parameter. The `By` locator for link with text is set with using `By.linkText(text)` and setting the text string as a parameter. When the element is found the method returns true, otherwise it will return false. The elements presence on the page is checked with the code shown in Figure 27.

```
private boolean isElementPresent(By by) {
      try {
            driver.findElement(by);
            return true;
      } catch (NoSuchElementException e) {
            return false;
      }
}
```

Figure 27. Method to check if the specific element is present and visible on the page

4.3.4 Summary of the authentication scheme test

On the one hand automating a web application with user certificate based authentication testing with Selenium is manageable. There is no need to disable authentication for testing phase and this enables running the application for testing also securely. Moreover, after creating a custom Firefox profile once for testing the test code itself is rather simple and the thing required for it is to initiate Firefox with that profile.

On the other hand this testing method forces creating multiple profiles to test different user certificates. This may be necessary if various users have different rights and authorization should be tested in miscellaneous situations. Although it is not especially difficult to manage multiple profiles it requires extra time and work from testers. Similar to HTTP Basic authentication testing some security measures in testing the user certificate based authentication are bypassed. As Selenium is not able to see security popups or pages displayed by the browser and this forces testers to find ways to bypass them. At least no user credentials are written into the code in plain text.

Lastly, this approach is not a secure way to manage testing. As there is a need to bypass many security restrictions, a lot of browsers inner security measures are turned off. For example adding additional master password to protect the certificates and other user information stored in the browser's profile. While there are some security concerns then implementing user certificate based authentication testing with Selenium, it is still considered more secure than testing login form or HTTP Basic based authentication schemes.

4.3.5 Conclusion

To sum up, test automation with Selenium for applications with user certificate based authentication is manageable and through custom Firefox profiles also rather easy. Although, it requires bypassing some important security measures in the browser instance. As already mentioned in Section 4.1 one concern towards Selenium framework in that part is being able to interact with the browsers security popups. It would make testing authentication schemes and other browser security measures possible and enable detecting vulnerabilities or faults in security implementations with automated tests.

4.4 Estonian ID-card

4.4.1 Overview of the application under test

First of all, in order to test ID-card the machine running the tests needs an ID-card reader to be attached to it and special software to be installed. Without it the testing is impossible. For application under test the official website to test the ID-card is used. It requires authentication with ID-card and is accessible from the URL `http://www.sk.ee/tervitus/`. In addition before using ID-card with Firefox browser users have to ensure that `EstEID` Firefox plug-in has been installed and is enabled and also that Estonian ID-card authentication module is installed and enabled under Firefox Extensions.

Moreover, before the restricted page can be accessed the user must authenticate itself with a valid ID-card. Before this can be done the ID-card has to be inserted into the reader that in return has to be attached to the computer. When accessing the URL with Firefox the browser will first prompt a security popup asking the user to choose the person credentials trying to access the page. The popup displayed is shown in Figure 28.

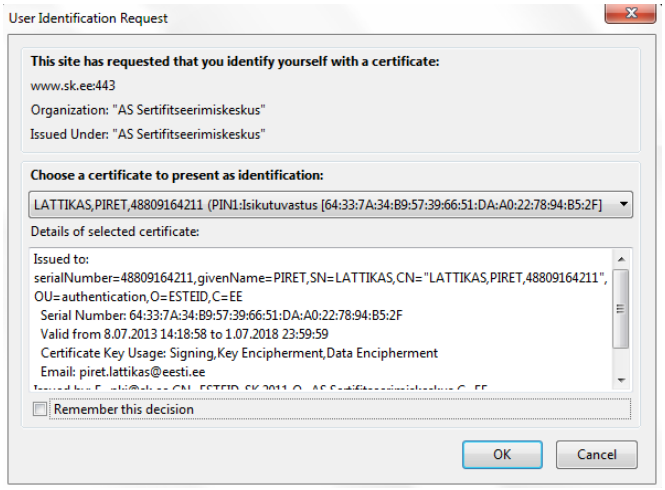

Figure 28. ID-card credentials security popup

After choosing the correct credentials the browser opens the second popup asking for the PIN code. The PIN code for authentication is four digits long and given to the user with the ID-card. After inserting the PIN code the authentication is successful and the user is directed to the restricted web page displayed in the Figure 29.

Figure 29. ID-card AUT web page

4.4.2 Preparations needed for the test implementation

Again the same problem occurs, there is no solution in Selenium to interact with browsers security popups. This is why there is a need to setup another Firefox profile for ID-card testing. As already mentioned previously the same Firefox profile cannot be used to test two different certificates. Following will present exact steps needed to configure ID-card certificate in the Firefox profile.

In addition as also covered previously Firefox Profile Manager is used to create a new profile. For the Selenium test to be able to access the created profile additional folder under the `src/test/` directory is created named `IDCardFFProfile`. Now again to add user certificate to the profile the browser is launched choosing the custom profile.

Similar to Subsection 4.3.2 it is needed to ensure that the browser will select the certificate for authentication automatically. Moreover, as demonstrated in Figure 16 the option "Select one automatically" has to be set on the "Certificates" tab on the Firefox browsers "Options" pane. Also it should be verified that necessary extensions and plugins have been enabled for the required profile.

Furthermore, ID-card certificate cannot be loaded like the user certificate from the tab "Your certificates". So to add ID-card authentication certificate to the browsers certificates list the AUT website URL is opened first. While, the browser does not display the certificate selection security popup anymore, what is asked is the PIN code for ID-card authentication. After inserting the correct code the user is directed to the AUT page

displayed in Figure 29. Lastly, then checking again the user can see his certificate listed under the "Your certificates" tab as noted in Figure 30.

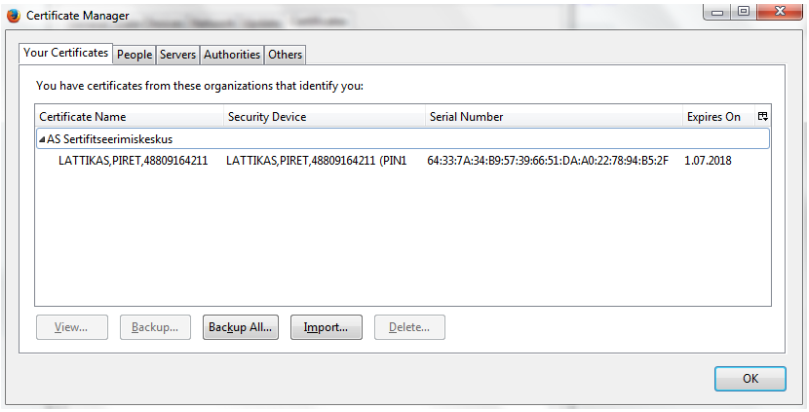

Figure 30. ID-card imported certificate

4.4.3 Implementation and execution of the Selenium test

Firstly, the URL specified for this test has already been mentioned previously. The test class used to automate the test for web application with ID-card based authentication is presented in the Appendix 7. Similar to the test for application with user certificate based authentication, the Firefox driver for this test is also initiated using the abstract class method `setUpFFDriverWithProfile` and the custom profile for ID-card testing created before is given in as the parameter. The code used for Firefox browser initiation is listed in Figure 31.

```
@BeforeClass
public static void createFFDriver() {
      // Launch the Firefox browser using Firefox driver.
      FirefoxProfile profile = new FirefoxProfile(new
           File("src/test/IDCardFFProfile"));
      setUpFFDriverWithProfile(profile);
}
```

Figure 31. Method to initiate Firefox browser with Estonian ID-card specific profile

Secondly as can be seen from the test class the JUnit method `EstonianIDCardTest` used for this test is not very long. As a first thing the URL for the application under test website is opened. From Figure 29 it can be seen that the page contains a link with text "www.id.ee". So again a method call `waitUntilLinkIsVisible` is added to ensure that the page is loaded before continuing with test execution. After the page has been loaded it can be seen that the website still asks for a PIN as demonstrated in Figure 32.

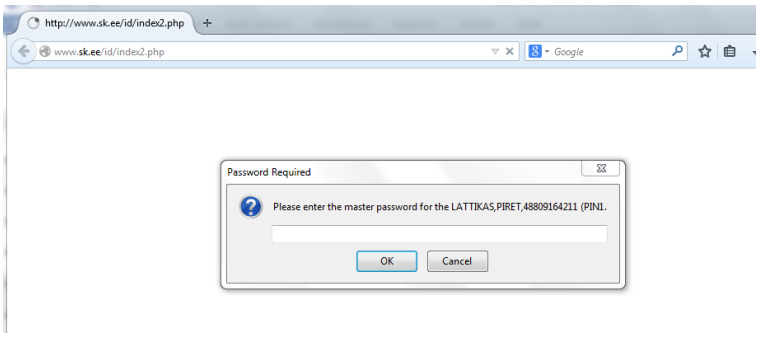

Figure 32. ID-card PIN popup

Moreover, for the test to pass at the moment the correct PIN is entered from the keyboard manually. After that the page is successfully loaded. But that test is not really fully automated and needs a user to interact by inserting the necessary code.

In addition, what is also noticeable on the Figure 32 is that the page is still loading meaning that even if the test would try to use Selenium commands to interact with the browser it would not be able to do so as the browser is still waiting for the PIN to proceed. To demonstrate that, a method in the abstract class named `takeAScreenshot` is added right after the call to the method `openUrl`. When adding a break point to the screenshot command and running the test in debug mode it is verified that the method is called, but no file is saved. The method used to capture a screenshot is demonstrated in the Figure 33.

```
protected void takeAScreenshot(String filename) {
      File srcFile = ((TakesScreenshot)driver)
           .getScreenshotAs(OutputType.FILE);
      File destFile = new File("src/test/Screenshots/" +
           filename);
      try {
            FileUtils.copyFile(srcFile, destFile);
      } catch (IOException e) {
            Assert.fail();
      }
}
```

Figure 33. Method used to capture a screenshot

Moreover, by adding breakpoint also to the first two lines of the `takeAScreenshot` method it is verified that the test execution hangs at the first line. This is when the driver is trying to interact with the browser to capture a screenshot from the page. Without human interaction the test will just hang and the browser will wait for the PIN until it is entered or some command terminates the execution of the test.

4.4.4 Summary of the authentication scheme test

While, test automation for applications with ID-card based authentication is possible, it cannot be fully automated. There is a need for a person to enter the PIN code, when asked for it. This in return means that there is no way to run these automated tests continuously for example as a Jenkins job without somebody monitoring the test execution on every run.

Furthermore, for these tests an ID-card reader with a specific inserted ID-card should be attached to the computer. The ID-card has to belong to the person whose ID-card was attached at the moment of saving the custom profile. Meaning that if different people wish to run the semi-automated test with their ID-card, they would have to follow the guidelines presented in Subsection 4.4.2 to set up their own custom Firefox profile.

This kind of testing raises security concerns. When testing is not possible through test automation to enable fast repeatable tests it may cause the tests not being run at all or being run rarely. Developers in a company that uses agile approach for programming are usually not keen in running tests that may run a long time or need some special setup. Therefore it may be that some vulnerability may occur due to lack of testing for applications with ID-card based authentication, which could have been discovered during testing phase.

Moreover, in the future Selenium framework developers should put more focus into making testing ID-card based authentication possible. Again it would be helpful if the Selenium WebDriver tool supported interaction with browser popup windows and this is a limitation in Selenium that should be addressed in the future. Enabling testing or making testing easier in different environments and with different users would also be helpful.

4.4.5 Conclusion

To sum up, test automation for applications with ID-card based authentication is not fully possible. The PIN code needs to be entered through the browsers security popup manually. This in return raises the security concerns as testing is probably not done as often, because it requires more resources. The future direction should be to enable or simplify test automation for the web applications with ID-card authentication.

4.5 Estonian Mobile-ID

4.5.1 Overview of the application under test

To demonstrate test automation for applications with Mobile-ID based authentication for application under test (AUT) a project under development was chosen. The AUT will run in a second machine, where it is developed and is accessible to the test from the URL `https://192.168.1.246:8080`. The application actually supports both ID-card based authentication and Mobile-ID based authentication. The login page displayed to the user contains the two buttons with corresponding images as shown in Figure 34.

Figure 34. Mobile-ID AUT Login Page Buttons

As explained in Section 3.5 for Mobile-ID authentication SOAP-based DigiDocService has to be used. As declared in the ID page section "Testing the services" the test DigidocService is available on the URL `https://www.openxades.org:9443/` [8]. In the development environment of the project exactly this URL is used for testing with Mobile-ID based authentication. The test site offers possibility to upload your own Mobile-ID certificates for testing or use one of the test numbers that do not require the presence of the actual physical mobile phone.

Secondly, the Mobile-ID button is pressed and a popup dialog opens asking for a phone number. Not to use a physical mobile phone for PIN entering a test number "00007" is typed to the field and the button named "*Saada teade*" is pressed on the form. After that a process bar is shown to the user showing the control code, which the user, in case of using real mobile phone, has to validate against the value displayed on his phone before entering the PIN for authentication. The time given to the user before the authentication process is cancelled by the system is two minutes. The progress bar displayed with the control code is presented in Figure 35.

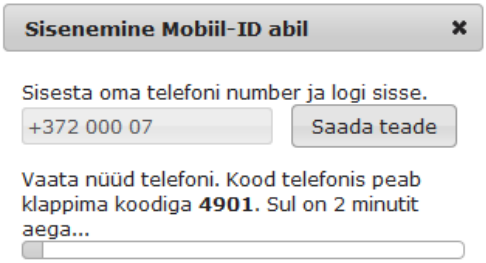

Figure 35. Mobile-ID authentication progress bar

Lastly, after that the test service for mobile-ID authentication sends a positive response verifying that the user entered the correct PIN code and is successfully authenticated. The system then authorizes the user giving him rights to access the application and directs the user to the main page of the AUT. On the upper right corner of the page the user name retrieved from the certificate and a logout link is displayed.

4.5.2 Implementation and execution of the Selenium test

The default Firefox profile will be used for this test and no additional preparations are needed. The test class created for test automation for application with mobile-ID based authentication is listed in Appendix 8. The URL shown in the previous subsection is used for this test.

Firstly, as with every test covered in this thesis the AUT page is opened. The page contains two buttons as shown in Figure 28 to choose the authentication method from. Before choosing and clicking the button for mobile-ID based authentication the test verifies that the button actually exists on the page. For this the abstract class method `isMobileIDButtonVisible` is called and the value returned is checked with JUnit method `assertTrue`. The method called is pointed out in the Figure 36.

```
protected boolean isMobileIDButtonVisible() {
      return isElementPresent(By
            .xpath("//a[@title='Mobiil-ID']"));
}
```

Figure 36. Method used to check if mobile-ID button is visible on the page.

Secondly, the Mobile-ID button is located using the same `By` locator used to verify that the button is present on the page. After it has been located it is clicked using the Selenium

method `WebElement.click()`. This in return opens the dialog box with the field to enter the mobile number used for authentication. So the number used for testing with value "00007" is entered to the field by calling the method `typeValueToNumberField`, which is listed in the Figure 37.

```
protected void typeValueToNumberField(String number) {
      WebElement field = driver.findElement(By.id("mobilenr"));
      field.clear();
      field.sendKeys(number);
}
```

Figure 37. Method used to type specified value to mobile number field

After this code has been executed the dialog box displayed to the user is demonstrated in Figure 38.

Figure 38. Mobile-ID authentication dialog box

Thirdly, the button "*Saaada teade"* visible in the figure is clicked by calling the abstract class method `clickOnNumberSubmitButton`, which will locate the button based on the identifier assigned to the element with value `btn`. As mentioned earlier the main page of the AUT contains a logout link. In the case of mobile-ID based authentication it is important to add the call to the method `waitUntilLinkIsVisible`. Otherwise the test execution will continue and the next assert will fail as the authentication has not yet returned with successful result and the page title is wrong. For this reason the wait command ensures the authentication process has finished before checking the main page title.

Lastly, to ensure that the authentication was successful it is needed to check that the page shown is the one expected, which in the case of this AUT is the main page. For this it is asserted that the title of the page is equal to the value "Main page". Moreover, it is known that the page displays a link with the text containing the user name value returned from the web service. Because of that it is verified that the page contains a link with the text

value equal to “SEITSMES TESTNUMBER”. The main page upper right corner of the AUT with the links displayed to the user is show in Figure 39.

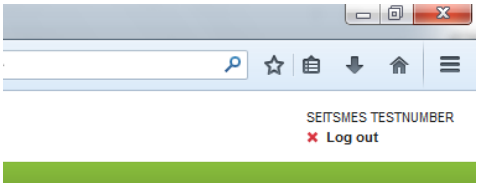

Figure 39. Mobile-ID AUT main page links

4.5.3 Summary of the authentication scheme test

On the one hand this way it is not possible to fully automate the test by using the real DigiDocService web service used for mobile-ID authentication. As testing the actual service requires a physical mobile phone and interference of a person. The person has to insert the PIN code on the phone for the authentication to be successful.

On the other hand with the use of DigiDocService test web service the test automation is possible and can be quite easily achieved. This can be done by just inserting a test mobile number on the mobile-ID authentication number field, which will return an expected answer from the test DigiDocService web service for authentication. In addition there are test numbers for different results. This enables test automation for situations where the mobile-ID certificate has expired or the user does not even own a valid certificate for authentication. So the developers are able to verify that in case of an error result the application would terminate the authentication process and restrict the user from seeing the protected pages.

Lastly, as already mentioned in Section 3.5 the mobile-ID has its advantages over the ID-card based authentication. This is also the case with test automation as full test automation for applications with ID-card based authentication is not possible. So it could be said that in overall the mobile-ID based authentication seems as a more secure authentication scheme than others discussed in this thesis.

4.5.4 Conclusion

To sum up, test automation for applications with mobile-ID based authentication is easily achievable. Using the test numbers enables testing of mobile-ID based authentication in different situations. Overall this authentication scheme has advantages in every aspect over other authentication schemes looked at in this thesis. In addition it is possible to test it and add automated tests to verify its reliability in different versions after changes have been implemented.

Summary

Most web applications use some sort of authentication to verify the user accessing the protected website. Running automated tests after making changes can help assure that important functionalities covered with automated tests did not break during the development. A popular tool for web application test automation is Selenium. In this thesis Selenium WebDriver tool was used to automate tests for web applications with authentication.

The goal of this thesis is to analyze the most common authentication schemes used in web applications and the possibility of automating the use of these methods in automated tests with Selenium. As a result of this thesis a documentation has been provided with steps how to implement these automated tests and what preparations are needed for each. In addition this thesis has drawn attention to the deficiencies in this field of Selenium framework. Authentication methods considered in the scope of this thesis were HTTP Basic, user certificate, login form, Estonian ID-card and Estonian Mobile-ID.

As a result of this thesis it has been concluded that test automation for web applications with authentication is possible for most authentication schemes focused on. Although, in some cases additional steps are needed to be able to manage test automation with Selenium. Testing user certificate and Estonian ID-card based authentication needs custom Firefox profile setup before tests can be executed. The use of Estonian ID-card cannot be fully automated in Selenium tests. The reason for this is that Selenium is unable to interact with browsers security popups, which are used to ask for the PIN code for authentication. It has been concluded that Estonian Mobile-ID based authentication is superior compared to the other methods when taking into account its security properties and support for test automation.

To sum up, test automation is important from security point of view. Selenium WebDriver is a good tool to use for test automation for web applications with authentication. Although it lacks the ability to interact with the browsers security popups making testing some authentication schemes and security measures difficult or even impossible. Future research should be focused on improving that part of Selenium or to find other measures to make it possible to test these parts of the application behavior.

References

[1] Alex, B., Taylor, L. A Minimal <http> Configuration – *Spring Security Reference Documentation*. [WWW] http://docs.spring.io/spring-security/site/docs/3.0.x/reference/ns-config.html#ns-minimal (01.05.2014)

[2] Alex, B., Taylor, L., Winch, R. Session Management – *Spring Security Reference Documentation*. [Online] http://docs.spring.io/spring-security/site/docs/3.2.3.RELEASE/reference/htmlsingle/#session-mgmt (29.04.2014)

[3] Apache Maven. What is Maven? – *Maven homepage*. [WWW] http://maven.apache.org/what-is-maven.html (05.03.2014)

[4] AS Sertifitseerimiskeskus. DigiDoc teegid - üldinfo ja teekide reliisimise ajakava. [WWW] http://www.id.ee/index.php?id=35779 (02.05.2014)

[5] AS Sertifitseerimiskeskus. DigiDocService spetsifikatsioon. [WWW] http://www.sk.ee/upload/files/DigiDocService_spec_est.pdf (02.05.2014)

[6] AS Sertifitseerimiskeskus. Isikutuvastus ID-kaardi ja Mobiil-ID'ga. [WWW] http://www.id.ee/index.php?id=31045 (02.05.2014)

[7] AS Sertifitseerimiskeskus. Mis on Mobiil-ID? [WWW] http://mobiil.id.ee/mis-on-mobiil-id/ (02.05.2014)

[8] AS Sertifitseerimiskeskus. Teenuste testimine. [WWW] http://id.ee/index.php?id=30303 (10.05.2014)

[9] Berjon, R., Faulkner, S., Hickson, I., Leithead, T., Doyle Navara, E., O'Connor, E., Pfeiffer, S. The keygen element – *W3C HTML5 specification*, 2014. [WWW] http://www.w3.org/TR/html5/forms.html#the-keygen-element (25.05.2014)

[10] Clooke, R. Keyloggers Come to Smartphones – *Mobile Security News site*, 2013. [Online] http://www.mobilesecurity.com/articles/452-keyloggers-come-to-smartphones (02.05.2014)

[11] Dierks, T., Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2 – *IETF document*, 2008. [WWW] http://tools.ietf.org/html/rfc5246 (25.05.2014)

[12] Eclipse Foundation. Eclipse homepage. [WWW] http://www.eclipse.org/ (25.05.2014)

[13] Editorial Team at Exforsys. Types and Levels of Testing in Programming, 2006. [WWW] http://www.exforsys.com/tutorials/programming-concepts/types-and-levels-of-testing-in-programming.html (04.03.2014)

[14] Endler, D. Brute-force Exploitation of Web Application Session IDs – *iALERT White Paper*, 2001. [Online] http://www.cgisecurity.com/lib/sessionids.pdf (25.05.2014)

[15] Fewster, M., Graham, D. Software Test Automation : effective use of test execution tools. London [etc.] : Addison-Wesley, 1999.

[16] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L. HTTP Authentication: Basic and Digest Access Authentication – *IETF Documents*, 1999. [WWW] http://tools.ietf.org/html/rfc2617 (28.04.2014)

[17] Friedrich, C. File:Ssl handshake with two way authentication with certificates.png – *Wikipedia*. [WWW] http://en.wikipedia.org/wiki/File:Ssl_handshake_with_two_way_authentication_with_certificates.png (28.05.2014)

[18] Froglogic GmbH. Squish GUI Testing – *Squish homepage*. [WWW] http://www.froglogic.com/squish/gui-testing/ (22.05.2014)

[19] Infosüsteemide turve: II Turbe tehnoloogia. / Vello Hanson, Ahto Buldas, Tarvi Martens ... [jt.] Tallinn : Küberneetika, 1998.

[20] JUnit team. Assertions – *JUnit wiki*. [WWW] https://github.com/junit-team/junit/wiki/Assertions (04.03.2014)

[21] JUnit team. JUnit homepage. [WWW] http://junit.org/ (18.05.2014)

[22] Kawaguchi, K. Meet Jenkins – *Jenkins wiki page*. [WWW] https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins (04.05.2014)

[23] Kolšek, M. Session Fixation Vulnerability in Web-based Applications – *ACROS Security*, 2002. [WWW] http://www.acrossecurity.com/papers/session_fixation.pdf (25.05.2014)

[24] Koomen, T., Pol, M. Test Process Improvement : a practical step-by-step guide to structured testing. Harlow [etc.] : Addison-Wesley, 1999.

[25] Laasik, H. Ainulaadne mobiil-ID – *Eesti infoühiskonna aastaraamat 2011/2012*. [Online] http://www.riso.ee/et/content/ainulaadne-mobiil-id (02.05.2014)

[26] Markvardt, M. Testimise automatiseerimine : mis see on (ja ei ole)? [WWW] http://cs.ttu.ee/tiki-download_wiki_attachment.php?attId=501 (04.03.2014)

[27] Microsoft. Maintaining Session State with Cookies – *MSDN Library*. [WWW] http://msdn.microsoft.com/en-us/library/ms526029(v=vs.90).aspx (29.04.2014)

[28] Mitt, I. Eestis on suurenenud internetikaubanduse kasutajate arv, 2012. [WWW] http://www.eestipank.ee/press/eestis-suurenenud-internetikaubanduse-kasutajate-arv-25072012 (04.03.2014)

[29] Mozilla organisation. Use the Profile Manager to create and remove Firefox profiles – *Mozilla Firefox support page.* [WWW] https://support.mozilla.org/en-US/kb/profile-manager-create-and-remove-firefox-profiles (04.05.2014)

[30] MTÜ Arvutikaitse. ID-kaart – *Arvutikaitse koduleht*. [WWW] http://www.arvutikaitse.ee/arvutikaitse-algtoed/id-kaart/ (02.05.2014)

[31] Odin Technology Ltd. Axe Test Automation Platform – *Products*. [WWW] http://www.axetest.com/products.html (28.05.2014)

[32] OWASP Foundation. Cross-site Scripting (XSS) – *OWASP website*. [WWW] https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (25.05.2014)

[33] Pällin, M. Automaattestimisvahendite kasutus ning praktiline ülevaade Seleniumi näitel : bakalaureusetöö. Tartu, Tartu Ülikool, 2012.

[34] Raghuvanshi, L. What is cookie? Advantages and disadvantages of cookies – *Webcodeexpert.com Blog*, 2013. http://www.webcodeexpert.com/2013/03/what-is-cookie-advantages-and.html (29.04.2014)

[35] Ranorex GmbH. Automated GUI Testing Tools – *Ranorex homepage*. [WWW] http://www.ranorex.com/automated-gui-testing-tools.html (22.05.2014)

[36] Seapine Software, Inc. QA Wizard Pro | Automated Testing – *QA Wizard homepage*. [WWW] http://www.qawizard.com/software-testing-tools/automated-testing/ (28.05.2014)

[37] Selenium Project. Getting Involved – *Selenium homepage*. [WWW] http://docs.seleniumhq.org/about/getting-involved.jsp (07.03.2014)

[38] Selenium Project. Introduction – *Selenium documentation*. [WWW] http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp (07.03.2014)

[39] Selenium Project. Selenium 1 (Selenium RC) – *Selenium documentation.* [WWW] http://docs.seleniumhq.org/docs/05_selenium_rc.jsp (23.03.2014)

[40] Selenium Project. Selenium homepage. [WWW] http://docs.seleniumhq.org/ (04.03.2014)

[41] Selenium Project. Selenium WebDriver – *Selenium documentation.* [WWW] http://docs.seleniumhq.org/docs/03_webdriver.jsp (04.04.2014)

[42] Selenium Project. Selenium-Grid – *Selenium documentation.* [WWW] http://docs.seleniumhq.org/docs/07_selenium_grid.jsp (04.04.2014)

[43] Selenium Project. Selenium-IDE – *Selenium documentation.* [WWW] http://docs.seleniumhq.org/docs/02_selenium_ide.jsp (07.03.2014)

[44] Selenium Project. WebDriver: Advanced Usage – *Selenium documentation.* [WWW] http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp (27.04.2014)

[45] Siles, R. Session Management Cheat Sheet – *OWASP website*. [WWW] https://www.owasp.org/index.php/Session_Management_Cheat_Sheet (29.04.2014)

[46] Simtec Limited. HTTP Authentication – *HTTP Gallery.* [WWW] http://www.httpwatch.com/httpgallery/authentication/ (28.04.2014)

[47] Skoudis, E. How should application developers manage cookies? – *TechTarget SearchSecurity website*, 2008. [WWW] http://searchsecurity.techtarget.com/answer/How-should-application-developers-manage-cookies (29.04.2014)

[48] Sourceforge users. Android GUITAR – *Sourceforge media wiki* [WWW] http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_Guitar (28.05.2014)

[49] Tähis, H. Autentimine ja identifitseerimine. [WWW] http://heiki.tpt.edu.ee/opiobjektid/turvaohud/autentimine_ja_identifitseerimine.html (04.03.2014)

[50] The Apache Software Foundation. Welcome to Apache Maven – *Apache Maven Project website*. [WWW] http://maven.apache.org/ (29.05.2014)

[51] The Spring Team. Spring Security. [WWW] http://projects.spring.io/spring-security/ (01.05.2014)

[52] WatiN Team. WatiN homepage. [WWW] http://watin.org/ (25.05.2014)

[53] Watir Team. Watir homepage. [WWW] http://watir.com/ (25.05.2014)

Appendix 1. Maven configuration

```
<project xmlns=http://maven.apache.org/POM/4.0.0
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>ee.lattikas.piret</groupId>
      <artifactId>thesis</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <packaging>jar</packaging>

.......

      <dependencies>
            <dependency>
                  <groupId>org.seleniumhq.selenium</groupId>
                  <artifactId>selenium-java</artifactId>
                  <version>2.40.0</version>
            </dependency>

            <dependency>
                  <groupId>junit</groupId>
                  <artifactId>junit</artifactId>
                  <version>4.11</version>
                  <scope>test</scope>
            </dependency>
      </dependencies>

.......

</project>
```

Appendix 2. Abstract Selenium test class

```
public abstract class AbstractSeleniumITest {
      private static WebDriver driver;

      protected static void setUpFFDriver() {
            driver = new FirefoxDriver();
            driver.manage().timeouts().implicitlyWait(new Long("10"),
                TimeUnit.SECONDS);
      }
      protected static void setUpFFDriverWithProfile(FirefoxProfile
          profile) {
            driver = new FirefoxDriver(profile);
      }
      protected static void quitBrowser() {
            driver.quit();
      }
      protected void openUrl(String url) {
            driver.get(url);
      }
      // Getting different values from the page
      protected String getPageTitle() {
            return driver.getTitle();
      }
      protected String getHeadingText(String heading) {
            return driver.findElement(By
                  .cssSelector(heading)).getText();
      }
      protected String getErrorMessage() {
            return driver.findElement(By.className("error")).getText();
      }
      // Checking if specific elements are present
      protected boolean isSearchFieldVisible() {
            return isElementPresent(By.id("search-text"));
      }
      protected boolean isSearchButtonVisible() {
            return isElementPresent(By.className("img-btn"));
      }
      protected boolean isLinkWithTextVisible(String text) {
            return isElementPresent(By.linkText(text));
      }
      protected boolean isUsernameFieldVisible() {
            return isElementPresent(By.name("username"));
      }
      protected boolean isPasswordFieldVisible() {
            return isElementPresent(By.name("password"));
      }
      protected boolean isMobileIDButtonVisible() {
            return isElementPresent(By
                  .xpath("//a[@title='Mobiil-ID']"));
      }
      private boolean isElementPresent(By by) {
            try {
                  driver.findElement(by);
                  return true;
            } catch (NoSuchElementException e) {
                  return false;
            }
      }
```

```
	// Interact with page elements
	protected void clickOnSearchButton() {
		driver.findElement(By.className("img-btn")).click();
	}
	protected void clickOnSubmitButton() {
		driver.findElement(By.name("submit")).click();
	}
	protected void clickOnTheMobileIDButton() {
		driver.findElement(By
			.xpath("//a[@title='Mobiil-ID']")).click();;
	}
	protected void clickOnNumberSubmitButton() {
		driver.findElement(By.id("btn")).click();
	}
	protected void typeValueInSearchField(String value) {
		WebElement field = driver.findElement(By
			.id("search-text"));
		field.clear();
		field.sendKeys(value);
	}
	protected void typeValueInUsernameField(String username) {
		WebElement field = driver.findElement(By.name("username"));
		field.clear();
		field.sendKeys(username);
	}
	protected void typeValueInPasswordField(String password) {
		WebElement field = driver.findElement(By.name("password"));
		field.clear();
		field.sendKeys(password);
	}
	protected void typeValueToNumberField(String number) {
		WebElement field = driver.findElement(By.id("mobilenr"));
		field.clear();
		field.sendKeys(number);
	}
	// Wait commands
	protected void waitUntilLinkIsVisible(final String linkText) {
		(new WebDriverWait(driver, 30)).until(ExpectedConditions
			.visibilityOfElementLocated(By.linkText(linkText)));
	}
	protected void waitUntilHeadingIsVisible(final String heading) {
		(new WebDriverWait(driver, 30)).until(ExpectedConditions
			.textToBePresentInElementLocated(By
				.cssSelector("h1"), heading));
	}
	// Take a screenshot of the page displayed
	protected void takeAScreenshot(String filename) {
		File srcFile = ((TakesScreenshot)driver)
			.getScreenshotAs(OutputType.FILE);
		File destFile = new File("src/test/Screenshots/" +
			filename);
		try {
			FileUtils.copyFile(srcFile, destFile);
		} catch (IOException e) {
			Assert.fail();
		}
	}
}
```

Appendix 3. Example Selenium test class

```
public class InitialExampleSeleniumITest extends AbstractSeleniumITest
{

	private final String url = "http://www.ttu.ee/";

	@BeforeClass
	public static void createFFDriver() {
		// Launch the Firefox browser using Firefox driver.
		setUpFFDriver();
	}

	@Test
	public void exampleTest() {
		// Open URL and wait for it to load
		openUrl(url);
		waitUntilLinkIsVisible("Tudengile");
		// Verify that opened URL has correct title.
		assertEquals("Tallinna Tehnikaülikool - Sinu elustiil!",
			getPageTitle());
		// Before searching verify that the search box exists.
		// Then type to search field.
		assertTrue(isSearchFieldVisible());
		typeValueInSearchField("cyber security");
		// Verify that the search button is present and click it.
		// Then wait for search page to load.
		assertTrue(isSearchButtonVisible());
		clickOnSearchButton();
		waitUntilHeadingIsVisible("Otsing");
		// Verify that opened page has correct title start.
		assertTrue(getPageTitle().startsWith("Otsing"));
	}

	@AfterClass
	public static void cleanup() {
		// Close the firefox browser.
		quitBrowser();
	}
}
```

Appendix 4. HTTP Basic Selenium test class

```
public class HttpBasicSeleniumITest extends AbstractSeleniumITest {

	private final String url =
		"http://test:123456@localhost:8080/HTTPBasicTest/";

	@BeforeClass
	public static void createFFDriver() {
		// Launch the Firefox browser using Firefox driver.
		setUpFFDriver();
	}

	@Test
	public void httpBasicTest() {
		// Open URL and wait for it to load
		openUrl(url);
		// Wait until Logout link becomes visible
		waitUntilLinkIsVisible("Logout");
		// Make sure the page is the one expected
		assertEquals("HTTP Basic Test Page", getPageTitle());
		assertEquals("Username : test", getHeadingText("h3"));
	}

	@AfterClass
	public static void cleanup() {
		// Close the firefox browser.
		quitBrowser();
	}
}
```

Appendix 5. Login form Selenium test class

```
public class LoginFormSeleniumITest extends AbstractSeleniumITest {

	private final static String url =
		"http://localhost:8080/LoginFormTest/admin";

	@BeforeClass
	public static void createFFDriver() {
		// Launch the Firefox browser using Firefox driver.
		setUpFFDriver();
	}

	@Test
	public void loginFormTest() {
		// Open URL and wait for it to load
		openUrl(url);
		waitUntilHeadingIsVisible("Login Form");
		// Verify that opened base url has correct title.
		assertEquals("Login Page", getPageTitle());
		assertTrue(isUsernameFieldVisible());
		assertTrue(isPasswordFieldVisible());
		//Fill in the login form with wrong data and click submit
		typeValueInUsernameField("piret");
		typeValueInPasswordField("password");
		clickOnSubmitButton();
		//Verify that you are on the same page
		// and error message is displayed
		assertEquals("Login Page", getPageTitle());
		assertEquals("Invalid username and password!",
			getErrorMessage());
		//Fill in the login form with correct data
		// and click submit button
		typeValueInUsernameField("test");
		typeValueInPasswordField("123456");
		clickOnSubmitButton();
		waitUntilLinkIsVisible("Logout");
		// Verify that the opened page is the one expected.
		assertEquals("Login form based authentication!",
			getPageTitle());
		assertEquals("Message : This is protected page!",
			getHeadingText("h3"));
	}

	@AfterClass
	public static void cleanup() {
		// Close the firefox browser.
		quitBrowser();
	}
}
```

Appendix 6. User certificate Selenium test class

```
public class UserCertificateSeleniumITest extends
      AbstractSeleniumITest {

private final static String url = "https://office.zerotech.ee";

      @BeforeClass
      public static void createFFDriver() {
            // Launch the Firefox browser using Firefox driver.
            FirefoxProfile profile = new FirefoxProfile(new
                  File("src/test/FFProfile"));
            setUpFFDriverWithProfile(profile);
      }

      @Test
      public void userCertificateTest() {
            // Open URL and wait for it to load
            openUrl(url);
            waitUntilLinkIsVisible("Jenkins");
            // Make sure the page is the one expected
            assertEquals("Dashboard [Jenkins]", getPageTitle());
            assertTrue(isLinkWithTextVisible("New Job"));
      }

      @AfterClass
      public static void cleanup() {
            // Close the firefox browser.
            quitBrowser();
      }
}
```

Appendix 7. Estonian ID-card Selenium test class

```
public class EstonianIDCardSeleniumITest extends AbstractSeleniumITest
{

	private final static String url = "http://www.sk.ee/tervitus/";

	@BeforeClass
	public static void createFFDriver() {
		// Launch the Firefox browser using Firefox driver.
		FirefoxProfile profile = new FirefoxProfile(new
			File("src/test/IDCardFFProfile"));
		setUpFFDriverWithProfile(profile);
	}

	@Test
	public void EstonianIDCardTest() {
		// Open URL and wait for it to load
		openUrl(url);
		waitUntilLinkIsVisible("www.id.ee");
		// Make sure the page is the one expected
		assertEquals("AS Sertifitseerimiskeskuse tervitus",
			getPageTitle());
	}

	@AfterClass
	public static void cleanup() {
		// Close the firefox browser.
		quitBrowser();
	}
}
```

Appendix 8. Mobile-ID Selenium test class

```
public class MobileIDSeleniumITest extends AbstractSeleniumITest {

	private final static String url = "https://192.168.1.246:8080";

	@BeforeClass
	public static void createFFDriver() {
		// Launch the Firefox browser using Firefox driver.
		setUpFFDriver();
	}

	@Test
	public void MobileIDTest() {
		// Open URL and wait for it to load
		openUrl(url);
		waitUntilLinkIsVisible("Kasutusjuhend");
		// Verify that the opened page is the one expected.
		assertTrue(isMobileIDButtonVisible());
		// Click button and enter phone number to field
		// Click the button to submit number
		clickOnTheMobileIDButton();
		typeValueToNumberField("00007");
		clickOnNumberSubmitButton();
		waitUntilLinkIsVisible("Log out");
		// Verify that the opened page is the one expected.
		assertEquals("Main page", getPageTitle());
		assertTrue(isLinkWithTextVisible("SEITSMES TESTNUMBER"));
	}

	@AfterClass
	public static void cleanup() {
		// Close the firefox browser.
		quitBrowser();
	}
}
```