# PIV Card Authentication Documentation

**Ben Nordin**

This documentation goes through the steps to make Smart Card Authentication possible on a tomcat server. More specifically, this documentation will show how to authenticate clients with government PIV (Personal Identity Verification) cards. The application created will give two options for authentication. One is a regular username and password form, and the other is with an X.509 certificate. All required files (except for certificates) are also on the GitHub repository: https://github.com/username/repo

## Required Technologies

This authentication process can of course be done with a number of different technologies, but for this documentation, Tomcat will be used with Java Spring. Before beginning, the following technologies should be understood and prepared:

- Tomcat 9
- Java Spring
- Maven
- AWS EC2
- Route 53
- Let's Encrypt

## Prepare Java Spring Applications

In order to accomplish optional PIV Card authentication, two separate spring applications must be created and deployed onto Tomcat. The first application will contain everything except for the authentication with the PIV card. The second application will only contain PIV card authentication. Each application works with Spring MVC to accomplish their tasks.

### PIV-MAIN

1. Start a basic spring application on the initializer website (https://start.spring.io/)
2. Import project to IDE.
3. Add the following dependencies into the pom.xml file

```
<dependencies>

		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>jstl</artifactId>
		</dependency>
		<dependency>
			<groupId>org.apache.tomcat.embed</groupId>
			<artifactId>tomcat-embed-jasper</artifactId>
			<scope>provided</scope>
		</dependency>
		<dependency>
			<groupId>taglibs</groupId>
			<artifactId>standard</artifactId>
			<version>1.1.2</version>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-engine</artifactId>
				</exclusion>
			</exclusions>
		</dependency>
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-webmvc</artifactId>
		</dependency>

	</dependencies>
```

4. The application will consist of two pages. One will be a login page where the user can select between PIV Card authentication and a regular username and password. The second page will output the username or CN of the logged in user. To do this, we will use JSP views. Place the following files inside the /src/main/webapp/WEB-INF/pages directory.

user.jsp

```
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c" %>

<html>
<head>
	<title>X.509 Authentication Demo</title>
</head>
<body>
	<h1>Login Successful!</h1>

	<form action="/piv-main/user">
		Username: <c:out value="${username}" />
		<br>

		<input type="submit" value="Back" />
	</form>
</body>
</html>
```

## userlogin.jsp

```
<html>
<head>
<title>X.509 Authentication Demo</title>

</head>
<body>

        <h1>Login Page</h1>
        <form action="/piv-main/redirect" method="get">
                <input type="submit" value="PIV AUTH" />
        </form>
        <br>
        <form method="GET" action="/piv-main/userlogin">
                Username: <input type="text" name="username" />
                Password: <input type="password" name="password" />
                <br>
                <input type="submit" value="login"/>
        </form>
</body>
</html>
```

5. Next, we need two configuration files so that Tomcat knows where our .jsp files are. Store these in src/main/webapp/WEB-INF

## mvc-dispatcher-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
                    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
                    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
                    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd">
   <mvc:annotation-driven />
<context:component-scan base-package="com.pivmain"/>

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
         <property name="prefix">
             <value>/WEB-INF/pages/</value>
         </property>
         <property name="suffix">
             <value>.jsp</value>
         </property>
    </bean>
</beans>
```

## web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
        version="3.1">

        <display-name>Servlet Information Application</display-name>

        <servlet>
<servlet-name>mvc-dispatcher</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

        <servlet-mapping>
       <servlet-name>mvc-dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>



</web-app>
```

6. Since this example is fairly small, only a few Java files are required. The first is a configuration class file. This file tells spring where our JSP files are.

WebConfig.xml

```
@Configuration
@EnableWebMvc
@ComponentScan("com.pivmain")
public class WebConfig implements WebMvcConfigurer  {

        @Bean
        public ViewResolver configureViewResolver() {
                InternalResourceViewResolver viewResolve = new InternalResourceViewResolver();
                viewResolve.setPrefix("/WEB-INF/pages/");
                viewResolve.setViewClass(JstlView.class);
                viewResolve.setSuffix(".jsp");
                return viewResolve;
        }

        public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
                configurer.enable();
        }

}
```

7. The next file is our controller. The controller contains four simple mappings.
   a. /user returns the view userlogin
   b. /userlogin is what the submit button on our userlogin JSP hits. This sends the username entered in the textbox to the controller, then redirects to /userpage
   c. /userpage returns the view user with the username added to the model
   d. /redirect redirects the user to the PIV Card authentication application

   PivController also has a String authUrl which is the URL where the piv authentication application is stored.

PivController.java

```
@Controller
public class PivController {

	String authUrl = "https://<domain-name>:8443/piv-auth/";

	@GetMapping(value = "/user”)
	public String user() {
		return "userlogin";
	}

	@GetMapping(value = "/userlogin”)
	public String formLogin(String username, String password) {
		return "redirect:/userpage/" + username;
	}

	@GetMapping(value = "userpage/{username}")
	public String userPage(@PathVariable String username, Model model) {
		model.addAttribute(username);
		return "user";
	}

	@GetMapping(value = "/redirect")
	public void method(HttpServletResponse httpServletResponse) {
		httpServletResponse.setHeader("Location", authUrl);
		httpServletResponse.setStatus(302);
	}

}
```

8. Finally, we have the User POJO that stores the username and password of a user.

User.java

```
public class User {

	private String username;
	private String password;

	public User(String username, String password) {
		super();
		this.username = username;
		this.password = password;
	}

	public String getUsername() {
		return username;
	}

	public void setUsername(String username) {
		this.username = username;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

}
```

9. Once all files are constructed, run the application to ensure that everything works. The links will not work due to the application running on the embedded tomcat server and not the standalone server that we will be deploying on, but try hitting "/user".
10. One more useful component to add to this application is a special plugin to the pom.xml file. This plugin makes it much easier to deploy our applications to our tomcat server. All that's needed is to run ***mvn compile war:exploded*** and maven will place our exploded war file into a location of our choosing. I chose to put it directly into my webapps folder inside of tomcat, but you can make it go wherever you'd like.

```
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-war-plugin</artifactId>
	<configuration>
		<finalName>piv-main</finalName>
		<webappDirectory>C:/Tomcat9/webapps/piv-main</webappDirectory>
	</configuration>
</plugin>
```

**PIV-AUTH**

1. Start a basic spring application on the initializer website (https://start.spring.io/)
2. Import project to IDE.
3. Add the following dependency to the pom.xml file:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

4. This application only acts as a certificate extractor, therefore, the only file that we really need is the controller. All other configuration will happen in the tomcat settings. In this controller, the "/" endpoint will find the X.509 certificate provided by tomcat, then extract the CN or Common Name from the file. In this simple example, the controller will then redirect the user back to the main application with just the common name in hand. All of the certificate information is in the subjectDN variable, and thus can be used as you wish.

AuthController.java

```
@Controller
public class AuthController {
	String projectUrl = "https://<domain-name>.com:8443/piv-main/userpage/%s";

	@GetMapping(value = "/")
	public void auth(HttpServletResponse response) throws InvalidNameException {
		HttpServletRequest request = ((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes())
				.getRequest();
		X509Certificate[] certs = (X509Certificate[])
request.getAttribute("javax.servlet.request.X509Certificate");
		X509Certificate clientCert = certs[0];
		X500Principal subjectDN = clientCert.getSubjectX500Principal();
		LdapName ln = new LdapName(subjectDN.toString());
```

```
                String username = "";
                for (Rdn rdn : ln.getRdns()) {
                        if (rdn.getType().equalsIgnoreCase("CN")) {
                                username = (String) rdn.getValue();
                                break;
                        }
                }
                response.setHeader("Location", String.format(projectUrl, username));
                response.setStatus(302);
        }
}
```

5. This application cannot be fully tested quite yet because of the required tomcat configuration. If you run the project and hit the "/" endpoint, your server should throw a null pointer exception. This is because the application is trying to retrieve an X509 certificate, but tomcat has not provided one yet.

## Setting up the Tomcat Server

1. After downloading Tomcat 9 from https://tomcat.apache.org/download-90.cgi, locate the server.xml file in {CATALINE_BASE}/conf. In order to have X.509 authentication on just one of our applications, we must have two connectors. In this example, I use a non-SSL connector for the main application, and an SSL connector for the authentication application. The main application can be HTTP or HTTPS, but its required that the authentication application be HTTPS.

```
 <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443"/>
<Connector port="8443"
protocol="org.apache.coyote.http11.Http11NioProtocol"
maxThreads="150" SSLEnabled="true">
  <SSLHostConfig caCertificateFile="fcpca.crt"
certificateVerification="require" certificateVerificationDepth="10">
    <Certificate certificateFile="cert.pem"
                 certificateKeyFile="privkey.pem"
                 certificateChainFile="chain.pem" />
  </SSLHostConfig>
</Connector>
```

The first connector should already be created in the file. A few notable files:

1. "caCertificateFile" is the Certificate Authority that the server will be used to trust an incoming X.509 certificate. The certificate must be signed using this private key or any of the subordinate certificates in order to be trusted by the server. In my case, I use the Federal Common Policy CA (https://fpki.idmanagement.gov/crls/), which is the root to all Government certificates, but this can be changed out for any.
2. The three files in the certificate tag are generated by Let's Encrypt. Leave these files as the same names and we will generate them later.
3. Tomcat reads these files from the CATALINA_BASE directory, not the /conf directory

2. The only thing left to do is to add the applications to Tomcat. With the plugin that we created earlier, we can call *mvn compile war:exploded* in both home directories of the application and the files should be copied directly over into /webapps.
3. To startup the server, navigate in a command prompt to the /bin folder and type in *startup*.
4. Again, the amount of testing that you can do is still limited. What you should expect to see is the login page at /piv-main/user. The authentication application, on the other hand, cannot be used yet because we have not generated our certificates yet. Hitting that page (/piv-auth/) on the 8080 connector should return a 500 error for the null pointer exception that we saw earlier. The 8443 connector will be unavailable until we set up the certificates.

## Launching an AWS EC2 Instance

Amazon Linux is a great operating system to use for deploying your applications to the cloud; however, Let's Encrypt does not support the OS. In this example I will use Ubuntu.

1. Log in to your AWS account and navigate to the EC2 console.
2. Select Instances, then Launch Instances.
3. For AMI, choose Ubuntu Server 20.04 LTS (HVM), SSD Volume Type.
4. You can choose from a wide array of instance types. I found that t2.micro (free tier) was not powerful enough to take a few simple requests so I upgraded to t2.medium and it has worked flawlessly thus far.
5. Select Next until you get to Step 6: Configure Security Group.
6. Select a premade security group or create a new one. The ports that we need to be open for incoming traffic are 8080, 8443, and 22. 22 is so we can SSH into the instance.
7. Select Review and Launch, then Launch, and you will be prompted to select a key pair to SSH into the instance. Create a new one or select an existing pair. If you choose to create a new one, a .pem will download from your browser. Save this for later.
8. Select Launch Instances. This will take some time. The instance should show up under instances with the name "-". I chose to name the instance "piv-test"
9. Finally, in the Instance Summary, under public IPv4 DNS, copy and save the shown address.

## Routing Domain Name to our Instance Using Route 53

The best way to route your domain name to your ec2 instance is by use of Route 53. If you are using a different service for your domain name, see their tutorials.

1. Navigate to Route 53 in AWS
2. Select Hosted zones, then the name of your domain.
3. Create a new Record Set.

4. For value, locate the Public IPv4 address (not the one from earlier) and paste it here.
5. After about 60 seconds, the AWS servers will update and the domain name will now be routed to your instance.

## Connecting to our Instance Using Putty

Feel free to use any SSH tool, I used putty because Windows does not ship with a built in SSH tool (yet) in command prompt. PuTTY does not accept .pem files for keys, so we must convert our file from before with PuTTYgen.

1. Download PuTTYgen at https://www.puttygen.com/
   Download PuTTY at https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
2. In PuTTYgen, load the .pem file from before, then save as private key.
3. Open up PuTTY and under Host Name, print ubuntu@<ec2 IPv4 DNS> where <ec2 IPv4 DNS> is equivalent to the address that we saved for piv-test
4. On the left-hand side under connection, select the plus symbol next to SSH, then click on the Auth word. Under Private key file for authentication, select browse and locate your key from PuTTYgen
5. Select Open, and you will now be inside your instance.

## Downloading Tomcat and Creating a Tomcat Service

Once inside our instance, we will need to download Tomcat and create a Tomcat Service. Because there are already great tutorials out there for doing this, I will just link the one I used: https://www.digitalocean.com/community/tutorials/install-tomcat-9-ubuntu-1804 Steps 1 - 5 are all that are necessary for our uses.

## Using Let's Encrypt to Generate Certificates

Let's Encrypt uses an application called Cert Bot. This must be download first.

1. Run the following commands:
   $ sudo su
   *# apt-get install software-properties-common*
   *# add-apt-repository ppa:certbot/certbot*
   *# apt-get update*
   *# apt-get install certbot*
   *# certbot certonly --standalone -d <domain name>.com*

2. That's it. Certbot has now downloaded a the files that we originally placed in our Tomcat server.xml file into /etc/letsencrypt/live/<domain name>.com/.
3. The last step is to copy those files into our /opt/tomcat directory where our server.xml file is expecting them.
4. Run the following commands (still from root)
   *# cp /etc/letsencrypt/live/<domain name>.com/cert.pem /opt/tomcat*

*# cp /etc/letsencrypt/live/<domain name>.com/chain.pem /opt/tomcat*
*# cp /etc/letsencrypt/live/<domain name>.com/privkey.pem /opt/tomcat*
*# chown tomcat:tomcat *.pem (make sure you are in the /opt/tomcat directory)*

## Deploy Files to our Instance

This is the final section to run our application. There are multiple ways to move our files from our local machine to our Ubuntu instance. One of the simplest ways is to push our files to a GitHub repository locally, then pull into our instance. This works especially well since git is already installed in our instance.

1. First, push your files into a repository on GitHub.com. Alternatively, you can just use the repository we have provided. (Don't forget to include fcpca.crt or whichever
2. Clone into your new repository:
   *# sudo git clone https://github.com/<username>/<repositoryname>*
3. Copy files to required locations:
   *# cp piv-main /opt/tomcat/webapps*
   *# cp piv-auth /opt/tomcat/webapps*
   *# cp server.xml /opt/tomcat/conf*

4. That's it! Just run a quick *sudo systemctl restart tomcat* and you should be good to go!