JOHANNES KEPLER
UNIVERSITY LINZ

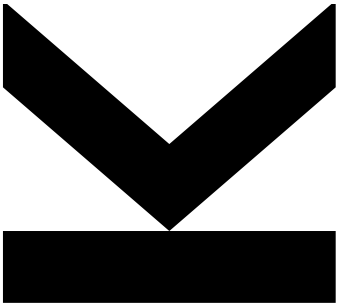
Submitted by
Patrick Schöppl

Submitted at
Institute of Networks and Security

Supervisor
René Mayrhofer

11 2019

Personal Agent Prototype in Rust



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Contents

List of Figures

List of Tables

Abstract

The so called Digidow Project aims to provide a decentralized solution for digital identity management. A key feature is to provide a service for authentication along with the identification of individual persons based on biometric features.
In the center of this idea a so called personal agent should provide this decentralized functionality for each individual user. The sensitive nature of the data this agent handles requires a special level of security standards on both the implementation and surrounding system.
This master thesis evaluates the programming language Rust as potential platform choice for the personal agent. We discuss the features Rust has been chosen for and which additional frameworks where selected and used to create the prototype we used for the evaluation. Furthermore, we dive into details about our prototype and present the implemented concepts. Moreover, we test our implementation and discuss our achievements, like isolated access to the hard drive, the developed concept behind the architecture and how incoming data is verified. Finally, we are going to discuss how future work can build on the introduced and existing concepts.

Abstract

Das Projekt mit dem Namen Digidow hat das Ziel, eine dezentrale Lösung für Identitätsmanagement bereitzustellen. Kernfunktionen sind die Authentifizierung und Identifizierung von einzellnen Personen, anhand ihrer biometrischen Merkmalen.

Im Zentrum dieser Lösung steht ein sogenannter 'personenbezogener Agent' (Personal Agent). Dieser Agent ist für jeden Nutzer dezentral, auf einen vom Nutzer ausgewählten Server, verfügbar und bietet die oben genannte Funktionalitäten.
Aus den von Natur aus sehr sensiblen Daten, mit denen der Agent umgehen muss, ergeben sich spezielle Anforderungen an die Sicherheitsstandards der Implementierung und des umgebenden Systems.
Diese Masterarbeit hat das Ziel, die Programmiersprache Rust als mögliche Sprache für diese Anwendung zu evaluieren. Wir diskutieren die ausschlaggebenden Rust Funktonen und welche Bibliotheken wir ausgewählt haben, um einen Prototyp für diese Evaluierung zu erstellen. Weiters werden wir im Detail den erstellten Prototypen beleuchten und die implementierten Konzepte erörtern. Darüber hinaus testen wir den Prototypen und evaluieren welche Punkte erreicht wurden. Beispielsweise, die Taskisolation, die Architektur und die Handhabung von ankommenden Daten werden untersucht. Im letzten Schritt wird gezeigt welche Ideen nicht umgesetzt wurden und wie auf das bestehende Werk aufgebaut werden könnte.

1 Introduction

What is a digital identity? How do we use it today? How are we going to use it tomorrow? Today, nearly everybody has at least a few. A well known example is the debit card. It digitally stores information about a person that enables an ATM to identify this person and allows access to a bank account.
In our context here, identities are a collection of data that belongs to a subject. Practically everything can be a subject. This may be an organization, a software program, a machine or a human [41, p. 2]. We are going to focus on the latest, the human. Like a small shadow of an individual human, a digital identity is a collection of attributes describing this person or aspects that are connected to the person. The following are some attributes that describe parts of the subject, split into individual groups based on the definition published in the ISO24760 documents [12]:

- Information about the physical existence: address, biographical details, location, date of birth
- Evolution over time: awards, degrees, qualifications
- Inherent to the physical existence: biometric
- Assigned: title, role, digital signature, social security number, passport number, cryptographic key
- Represents identity: passport, business card

In our daily life we experience that such an identity may be used in many scenarios. This could be to verify that we are allowed to access and use a wanted resource or perform a certain task. When buying some special goods, a person may have to provide an identity that verifies the age of the potential buyer, for example a drivers licence. The driver licence provides properties that allow a vendor, in the role of an security authority, to determine the validity of the credential by checking if it looks real. Furthermore, it is possible to use biometric information, the picture, to verify that the identity belongs to the person who claims to own it. If the licence is valid and the person looks like the one on the picture, the attribute (age) is checked. Based on this information the vendor can decide if the person is allowed to buy what he wants [41, p. 2.2].
The same process can be projected to many areas in our daily lives. Moreover, it can be transferred into a digital version of the above story. Maybe, the vendor is equipped with a special scanner, that allows, based one an inherent biometric feature, to access a digital identity that verifies the age of the potential buyer. Something similar, but without biometric is already implemented when buying cigarettes in Austria. The buyer has to insert its banking card on which the attribute age is stored. After all, opening a secured room with a fingerprint or unlocking a smart phone with a picture of the face is already reality. Both cases allow a user to do something, because he verified that an identity, that contains permissions to open or unlock, is owned by them.

1.1 Project Digidow - Digital Shadow

Let us look a few years into the future based on what we observe today. We assume that at some point, each identification and authentication is going to be done based on inherent body features. Such a feature could be used to unlock and access a stored digital identity which verifies that we are who we claim to be. Tied together with a certain certificate, we can verify that we are allowed to do, what we are trying to do.

In order to accomplish this on a global scale, it will be required, that a persons identity is stored somewhere in a global network. A scanner that is able to collect biometric features, sends the scanned data to a global well known authority. This authority may identify the person and create a connection to its digital identity. Afterwards, this authority can send the requested data for that specific user to an entity that wants to verify that the user is allowed to do what they are trying. Let us call this entity a verifier. This verifier can be a simple door lock to a flat, a border control or a police officers laptop.
The benefits are that everybody with an personal abler can not forget their ID card and the identification could be done much faster without any doubt that the person really is who they claim to be.
This process may be triggered on many occasions in the daily life. Opening a door, starting a car, entering a office building, entering a train and so on. Doing this with one single authority that can authenticate the user and stores all required data, may be very comfortable. However, this authority would be aware about everything a single user does, knowing all their movements. Furthermore, this authority also controls what this particular user can do or can not do. Bringing this to an extreme means, removing a person from the one single system, leads to a 'nobody' that may not be capable to interact with anything in a modern world. Furthermore, it may not be possible to control or stop this authority from identifying a user and therefore it will be impossible to enjoy anonymity in the public.
Preventing that this power gets into the hands of one single major player, is the self set target for Digidow [34] [7]. Solving this comes with a great price of complexity and many security related challenges.
At the end, it should be possible for every living person to own a so called personal agent. This personal agent is going to be the one and only entity that has access to biometric templates, and certificates. Most importantly, the agent is in the possession of the individual user. Therefore, the users can choose where to install or host their personal agent. Moreover, they can decide, monitor and configure what is happening with there identities, and are able to turn it off.
Digidow aims to provide such a solution.

1.2 Problem

We mentioned, a key element in the Digidow project is the personal agent. This agent is going to be the central point in the authentication process, with access to all certificates, ID's and biometric templates. Consequently, it is in the users interest to comply and enforce very high security standards.
It may be seen as the official digital copy of an individual person. Therefore, the software should be as secure and resistant to failure as possible. Any fail creates mistrust, and furthermore, it may also do immense and irreversible harm to an individual user. At the end of the day, the user of this personal agent has to be able to trust it with his digital life.

A consumer of this system has to trust the implemented and provided software. Furthermore, the underlining system (hardware) on which the software is installed. In this context we name this system a 'trusted system'. A trusted system is a system that verifiable follows common and current security standards, to protect itself and the users against any harm. To be able to guarantee a certain level of security upon users can build their trust on, it may be necessary to formally

verify each part of the system that should be trusted. This means, at least in an ideal world, each piece of hardware, each line of code is derived from a formal specification and the resulting implementation can be tested and verified against it.
However, in an infrastructure with complex and protected hardware, usable for modern operating systems it is not feasible to verify the whole system any more. In fact, the users have to give their trust to software and hardware, based on statements and promises that manufactures gave them. With that unpleasant circumstance in mind, it seams like an logical step to try to minimize the unverifiable parts of the system.
If we take a look on an traditional PC we can determine multiple layers that we have to trust. We have to trust the hardware and its firmware that is in our PC, notebook or smartphone. On top of the hardware there is the operating system. Usually, the hardware manufacturer decides which operating system should be preinstalled or supported on the device. Therefore, the manufacturer chooses the operating system that has to be trusted by the users. Finally, there are the applications that are running at the highest layer on top of the operating system.
Usually, everything besides the additional installed applications, is out of reach for users. This means, the system a user has to trust is already quite big, starting at the hardware going up to the operating system together with the drivers and default applications. It is nearly impossible to verify each part of the underlying systems. Therefore, we have to trust that:

- The firmware, driver, operating system and hardware is bug free and does not allow an attacker to infiltrate our system.
- There is no hidden loophole that allows agencies or manufactures to access data or circumstance security provisions.
- Provided security features are well designed and in fact secure.

Still, there are ways to reduce the uncertainty. When choosing open source solutions it is theoretically possible to verify the security of these implementations. However, this most likely will fail due to the complexity. Often, only small parts that seem to be security relevant are checked intensively by the community standing behind an open source project. Furthermore, there exists open source hardware, where the layout is public accessible [24]. Though, the project does not seem to be a huge success.
Another way to reduce the size of the trusted system, are unikernels. An application that runs as unikernal does not require an underlying operating system. Instead, the necessary hardware drivers and libraries are added and compiled directly into the application.
Nevertheless, the common way is increasing security measures in the top layer to mitigate security risk in the lower levels.

1.2.1 Application Layer

Many modern languages like C# or Java come with an underlying framework or similar. C# needs the .net Framework [31] and Java needs the Java Virtual Machine [3]. Working with languages that come with a layer underneath, provides many benefits in terms of speed, memory security, portability and so on. However, it also adds another layer that we most likely can not verify and that we have to trust, if we use it.
Removing that layer, in order to minimize the size of the system we have to trust, leads us to another kind of languages, namely system languages like C and C++. Both languages are well

known and broadly used. Looking at C++ today, reveals a modern and very complex language. Often it is the first choice when an application has to be very fast or if the developer needs the ability to manage system resources. Essential things, like memory management and type usage can be in the developers hand, leading to an extraordinary responsibility. Nearly nothing is impossible in these languages and this leads to an big error domain. Buffer overflows and leaking memory are just two common mistakes when C/C++ is used.
Therefore, this freedom can and most likely is going to introduce new attack vectors and new uncertainties that get harder to find and prevent with each line of code.
Considering the drawbacks that come with a system language together with the goal to minimize the attack surface, switching to such a language seems not to be a wise choice. Luckily, since a few years there is Rust as an alternative general purpose system language that solves some of these issues.

1.2.2 Rust

Rust is an object - functional system language, developed by Mozilla. There are a few major differences between Rust and C/C++ which make Rust a type safe and memory safe language [21]. Besides some distinctions on the syntactical layer, the compiler that comes with Rust is responsible for the difference. After compilation, the resulting binaries is very similar, as if we would have written the same application in C.
From the Rust homepage rust-lang.org we learn that Rust offers us many, partly common, features like:

- zero-cost abstraction
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- efficient C bindings

Two of the outstanding selling points are threading without data races and guaranteed memory safety. We are going to see in Capture 3.2 how Rust uses a ownership concept for variables, to be able to make such guaranties.
Rust is on a static way to gain more trust from developers and a language that the community seems to like. GitHub announced that Rust was the most loved language in 2016, 2017 and 2018 [17], and the amount of new Rust projects and developers is constantly rising. Anyway, this feature list combined with the a very good reputation and with a peak on ongoing projects leads us to the assumption that Rust might be the language of choice for the proposed personal agent.

1.3 Summary

Summing this up we can derive that our prototype should:

- Put as much control as possible into the hands of the developer, in order to allow a verification of as many parts of the resulting system as possible.
- Use a stable and secure environment, similar to what runtime based languages like Java, C# provide, to omit errors that are typical for system languages.

On the other hand, by analysing if a runtime based language suites our needs we found that:

- Because of the complexity that a runtime itself add to a system, another part of the resulting system gets unverifiable.
- A runtime like JVM or .NET may introduce there own vulnerability.
- The executed code may not be static and may change during execution.
- A runtime consumes CPU and memory.

Based on the drawbacks a runtime adds to our system, we decided that we are using a system language for the prototype. Furthermore, based on what we learned about Rust, we see that this language provides built in features that allow us to write code that omits many typical system language errors especially concerning the memory management. Rust therefore was selected because it provides security related features without introducing new complexity to the resulting system.
The goal is to implement a personal agent prototype as a proof of concept in Rust. It should show the basic functionality and evaluate if Rust is a suitable choice. Moreover, it shall follow a strict security thinking, in order to set a reasonable direction for further development.

1.4 Thesis Overview

The agent is going to act in a way similar to a digital identity provider. In Section 2.1 we take a look on various provider, protocols and ideas in this area. Also, isolation - a method to limit the influence an application can have on a host system, is an important part of this work and is introduced in Section 2.2. Before we go threw a detailed specification for the targeted implementation in Section 4 and 11.3 we discuss in detail why we choose Rust and furthermore take a look on some Rust features in Section 3.
After that we are going to reflect about the architecture, used frameworks and concepts behind the implementation before we finally look at details of the final system in Section 9. In Sections 10 we will recapitulate if the agent, meets the set goals. Before we conclude on our work, we also discuss possible steps, that may be done by future developer in Section 12.

2 Related Work

Before we dive into details about Rust and the implementation itself, we want to introduce two major topics that are strongly related with the ideas behind this implementation. In the first part we discuss identity provider, protocols and current ideas in this area. We are going to look at this, meanly because the targeted personal agent has to provide that kind of functionality and is going to function as an alternative in this area.
The second part introduces process isolation, an important focus during the development of our prototype. We are going to discuss a selection of technologies and explain on the real life example Google Chrome, how different methods can come together and why this is an efficient security measurement.

2.1 Digital Identity Provider and Protocols

Before we introduce some provider and discuss their service it is important to know what an identity provider in general offers and provides. Many services require that the user is identified and linked to an identity. This service could be an web-application like Google Plus, Instagram or a part of infrastructure in a company like Outlook. The straight forward solution is to have an account with username and password for each application. Each provider, manages, requests and stores its own desired properties about the user. For example name, age and access rights, may be stored. Because of the immense amount of services and applications that are in need of such a link to a profile, each individual person has a collection of digital representations. But all accounts belong and represent one single subject. Furthermore, there are many attributes, like the name, that are stored in multiple accounts redundant.
A simple solution would be to have one single identity, stored at one central point with all possible required attributes like for example name, age, workplace, address, partners. Services that are in need of a information about the user can send a request to this central point. This central point provides the data for these services and is called an identity provider. A service where the user can authenticate once and use this authentication and profile in multiple applications.
Depending on the provider, an application can request some public information (like the name), or can request an confirmation that this login represents an actual person.
Today we know the importance, range and power of such a service. Facebook, may be the biggest collection of digital identities world wide, and we experience what missing control can lead to. One scandal rushes after the other like the data breach from September 2018 where 29 million user details where stolen [39, 20].
Back in 2005, Windley Phillip published the book 'Digital Identity'. He experienced and foreboded the importance and value of digital identities. His goal was to set a direction towards a mindful handling with this data. Moreover, he wanted to introduce some best practice methods on how to manage such identities and to point out the importance of national and international legal control [41]. In the year 2016 the EU Parliament approved the General Data Protection Regulation (GDPR) after four years of preparation. The objective is to lay down rules for protection of natural persons, there rights and freedoms, with regard to the processing and free movement of personal data and their right to the protection of personal data [38, Art 1].
Windley suggested some questions that one should consider when it comes to services that collect and use personal data. Hereinafter, we are going to consolidate some of the suggestions from the

book with suitable articles from the General Data Protection Regulation.

- What kind of identity data are collected? (What?) Ideally, it is a limited collection with only necessary information.
  The GDPR defines that personal data shall be: 'adequate, relevant and limited to what is necessary in relation to the purposes for which they are processed ('data minimisation')' [38, Art 5.1.C].

  Furthermore, there are regulations in Article 9 to prohibited processing data considering special categories of personal data [38, Art 9]. Such special categories are among others, biometric data, religious or philosophical beliefs and sexual orientation.
- How is the identity data collected? (How?) Ideally, it is transparent and the user explicit hands over the information.
  In Article 13 and 14 the GDPR defines rights for the data subject to be informed where and where not personal data are collected [38, Art 13, 14].
- Why was the identity data collected? (Why?) Ideally, it is specified and documented why the data is collected.
  Article 5.1.b states that personal data has to be collected for specified, explicit and legitimate purposes and no further processed in a manner that is incompatible with those purpose [38, Art 5.1.b]. Followed up with the already mentioned Article 5.1.c, where a data minimisation is demanded, one can assume that the data collected is the minimal set of data that is required to use the service.
- Who uses the data? (Who uses?) Ideally, the data is only used for its predefined purpose and is stored as short as possible.
  Article 5.1.b requires the data collector that personal data shell only be 'collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes'. Furthermore, Article 5.1.e states that personal data should only permit identification of data subjects for as long as it is necessary [38, Art 5].
- Where is the data stored? (Where?) Ideally, the data is stored in an area with strict privacy laws. The user is at least aware where this is.
  The GDPR defines rules for personal data transfers into third countries or internal organisations. Article 44 demands that if data is transferred into countries, it has do be guaranteed that the level of protection of natural persons is not undermined [38, Art 44].
- Who is the data owner? (Who owns?) Ideally, each subject is able to access its data and is able to change and delete them.
  In Capture 3 the GDPR gives the subject rights, to access the data, a right to be forgotten, a right to transmit the data to another controller, and some other right considering there own personal data. [38, Art 15,17,20].

Summing this up, we see, that the mindset and required templates for such rules where defined even before the EU started to address these issues with the General Data Protection Regulation.

2.1.1 OAuth 2.0

Before we take a look at some provider, we discuss a basic protocol that enables a provider to authenticate and authorize users. Furthermore, and most importantly, that enables other applica-

tions to use the provided service.
OpenID is known as an authentication protocol and OAuth 1.0 as an authorization protocol. OAuth 2.0 combines both and offers authentication and authorization in one open protocol. Identity provider can implement this protocol to offer their service.
The target is to unify the required accounts for a user and offer a solution with one account and a single required login action.
OpenID Connect is built on top of OAuth 2.0 [25], but we are going to focus on OAuth 2.0. It is a broadly used protocol, with the maybe most prominent implementation by Google, Microsoft and Facebook [30, p. 91].
The following example shows the a possible routine for a login process. The client, for example a web application, redirects the user to the identity providers login page. The call contains the clients consumer key and a callback URL, to which the user is routed afterwards. On the authorization page, the user can authorize the service to access his information. After that the callback URL is called, together with a verification code. Now the client can query an access token, with the received verification code. If the client/service wants to query some user data, it has to include the access token in all provider calls.
One element of this process that helps to understand how OAuth2.0 is used, in respect of different types of clients is to take a look on grant types. OAuth2.0 offers four different grant types [30]:

- Authorization code grant type: Is recommended for applications that can spawn a web browser. The registered client application must redirect the user (resource owner) to the authorization server, in order to get approval. Figure 1 illustrates this process.
- Implicit grant type: Is mostly used by JavaScript Clients that run in the browser. One major difference is that the need for an authentication request is done implicitly. Similar to the previous method, the user will be redirected to the authorization server.
- Resource Owner Password Credentials grant type: The user must trust the application and has to give it is credentials directly to the client application.
- Client Credentials grant type: This is used if there is no user as resource owner instead it is only the client application that communicates with the authorization server.

Summing this up, OAuth2.0 is a broadly used protocol that allows authorization and data sharing between an identity provider and the client.

2.1.2 Persona.org

Persona was a Mozilla project with the goal to provide an easy to use, easy to implement solution for users to login into multiple websites. It was designed to replace OAuth1.0. The idea was to issue a certificate bound to an e-mail address of the registered user. Therefore, any service that offers e-mail addresses for its users, can become a persona identity provider. The new provider had to publish a page at '/.well-known/browserid' containing three things:

- Public-key: The public part of the domains cryptographic key.
- Authentication: The domains page for asking users to log in.
- Provisioning: The domains page for certifying its users' identities.

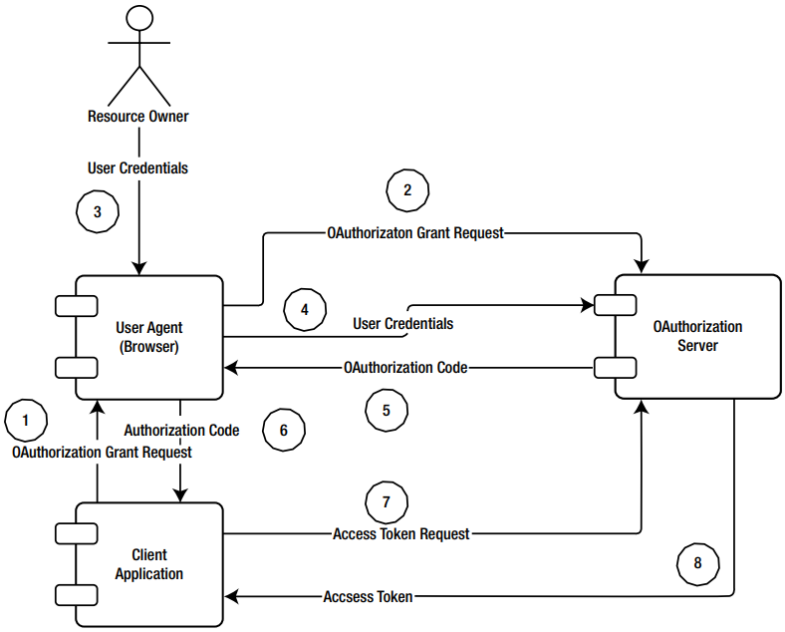


Figure 1: OAuth2.0 Authorization code grant type [30]

The following is an example, given by Mozilla, that allows a better understanding on how Persona.org works. Lets assume that example.com is a valid and working Persona identity provider. When Alice wants to visit a page the first time this will look like the following:

1. Alice's browser fetches a unique id for this session
   from https://example.com/.well-known/browserid.
2. Alice's browser invisibly loads the provisioning page for example.com and asks it to sign a public key certifying Alice's identity. The provisioning page is the domains page for certifying its users identities.
3. Before signing the key, example.com needs proof that the user is Alice. Until now, Alice did not Login into her account, therefore the provider needs an authentication from Alice. The browser shows Alice the authentication page for example.com, where she can signs with her username and password, and with that establishing a new session at example.com.
4. Alice's browser reloads the provisioning page and again asks it to sign a public key certifying Alice's identity.
5. The provisioning page can verify Alice's identity by inspecting the new session. Satisfied, it signs a certificate containing Alice's public key, her email address, and an expiration date for the certificate.

If she has already logged into another service, step 3 and 4 are skipped.
The benefit behind this system is that no data besides the public signature is shared between the third party page and the identity provider. Nevertheless, because of low usage the support for Persona ended on November 30, 2016 [26].

2.1.3 Windows Hello for Business

Another provider we take a look at is Windows Hello for Business. As the successor of Microsoft Passport, it allows to use multiple services with one single Microsoft account. The key focus for 'Windows Hello for Business', and reason why we mention this here, is that Microsoft wants to give the user an alternative for its login. It allows the windows user to replace the default passwort / username login with a more secure two factor authentication. Instead of the well known combination a mixture out of 'Device' + 'PIN' or 'Biometric' is used. The user combines something he has (the hardware) with something he knows (the pin) or if the hardware is available with a biometric feature as something that is part of the user. There are multiple reasons why this can be beneficial:

- Users tend to find it hard to remember strong passwords, reuse them on multiple sites, write them down.
- Server breaches may expose symmetric network credentials like passwords.
- Passwords are subject to replay attacks. This means, that the someone records for example the network traffic, and 'plays' it again when he wants to trick the server.
- A user may be targeted by phishing attacks, and expose the password.

The service is available for users with a Windows account, an active directory account, a Azure active directory account and any idenity providor that supports Fast ID Online (FIDO) v2.0. The FIDO Alliance follows the mission to develop technical specifications that define a set of mechanisms to reduce the reliance on password for user authentication [9].
Figure 2 illustrates the discussed process for the Microsoft solution.

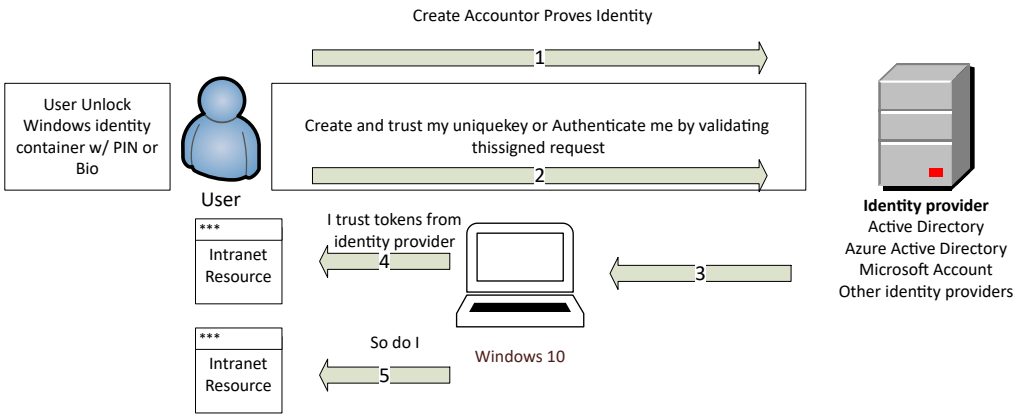


Figure 2: Windows Hello Authentication Flow

Source: https://docs.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/images/authflow.png

The generated authentication token is used to unlock the current device and can also be used to login into other services like web applications.
In detail, windows hello used a public, private key pair with which the credentials can be bound to the device. Also the generated authentication token is bound to the device, therefore the token can only be used there and stealing it would be meaningless. Furthermore, the process can make use of a Trusted Platform Module (TPM) to store the private key. With TPM it is possible that

the private key never leaves the device, leading to increased security. If the already mentioned authentication methods are used and TPM is available, the private key is locally unlocked and the bound authentication data are sent to the authentication server.
Depending on the solution, Microsoft certainly stores the public key, and if TPM is not available, also the private key. Biometric data, pins and other authentication method will only be stored on the device [42].

2.1.4 Facebook Connect

Facebook Connect allows a user with a Facebook account to use this account on third party services. The main benefit is that the user does not have to create a new account for each new web service. Furthermore, the web service gets a full trustworthy identity profile, without having to ask the users about there details.
The core for web application is a simple to integrate JavaScript library. This provided library enables well known services, for example the 'like' button and 'login with Facebook', for this web service. There are also libraries for Andorid and iOS to allow applications to login and connect with Facebook.
Today, Facebook Connect also uses OAuth2.0 to authenticate, generate a access token and share data. The platform supports three types of access tokens: user access tokens, app access tokens, and page access tokens.

- User access token: Most commonly used. Allows to access Facebook in behalves of the user.
- App access token: Used to update application settings.
- Page access token: Similar to the user access token, but also allows to retrieve a page token for the pages and apps that a person administers.

In order to get a better understanding how such a token is generated and used in the users perspective we outlines with Figure 3 the process of accessing from a web service for the first time: When the user uses Facebook to access a service, on the first visit, a pop-up will inform the user about the privileges the webservice wants to obtain. By default, all public information are accessible, this includes name, profile picture, age and gender. The main difference to most other identity provider is that Facebook has a big data collection about the user. The granted service, will have access to all public information that are linked with that account. In return, Facebook gets insights in user interests outside of Facebook. Furthermore, this information can be used for targeted advertisement.
The critics here are that data sharing is often not transparent for the user and that Facebooks collects data without the users understatement [23].

2.1.5 Blockchain Projects

Blockchains are contentiously growing digital lists of data blocks. These blocks are chained together based on a cryptographic procedure, where the validity of any new block is based on the previous one. Each of these blocks contain one or more transactions. A transaction can contain any kind of data, like money transactions, identity requests or verifications or for example user information. Furthermore, this chain is distributed and synchronized between multiple clients around the world. These clients may add transactions to the current block, or are just interested

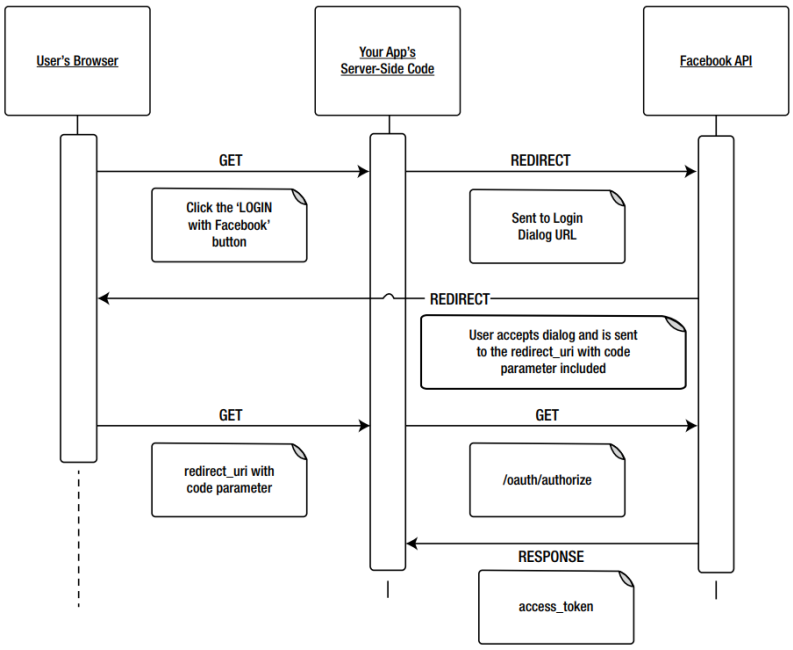


Figure 3: OAuth2.0 Facebook [30]

in what is happening in the chain.
Today, the goal of multiple projects is to store digital identities and access tokens in blockchains.

Blockstack The goal is a 'new' Internet, where a user owns his own identity. Each user owns a so called Blockstack ID that, based on the block chain technology, allows to create a secure mapping to usernames, public keys, and data storage URIs. An application may use the users permission and the Blockstack ID to access data. Furthermore, all applications are server-less and decentralized, running at the clients hardware, in this approach. The service can use the Blockstack identity and not have to care about other user management systems.
The user can chose between Dropbox, Google Drive and similar services, to store the actual data for each service. This data may also be encrypted and only the data URI is published and bound to the identity [2]. Therefore, the user has full control over his profile and its data.

ID2020 is a project with the goal to provide digital identities containing everything from birth certificate to driver licences and medical records. Especially in poor countries, the lack of an official identity is still a common problem, that may solved with this approach.
An official identity is often required to open a bank account, to access healthcare, education, vote and to be part of social assistance programs. Therefore, providing a solution for these people brings a benefit for both people and government. They offer a personal, persistent, private and portable way to manage these identities [11]. In short, the data is stored in a blockchain and can be accessed with biometric identification.

2.1.6 Summary

Summing up the inspected digital identity solutions we see that there is a need for identity management in the constantly growing digital world. The provider role gives a services the control

over collected identity information. A user has to relay on the goodwill of the provider that they do not abuse this power and that they provide there service when the user needs it.
Further decentralizing the identity management allows the user to gain back control over his identity. It also adds more reliability to the system by distributing the information all over the world, in a secure way. Projects like Blockstack and ID2020 already bring this benefits to some countries.

2.2 Sandbox

For our prototype we want to make use of sandboxing. In a nutshell, sandboxing is when we restrict the operations a process can use.
This is important, because there are certain instructions that are triggered or controlled from third parties and therefore they are not complete under our control. An example is reading a file based on a file name that was given by a third party like an user or an external device like a scanner. This interface could be used to purposely or without intend inject content into our system with the potential to harm it. Therefore, these operations are not trustworthy.
Our goal is to extract parts, where instructions or data influenced from the outside is processed, and put them in a so called sandbox. This sandbox hast the potential to limit the possibilities a program has by enforcing tailored rules on the process. Within the sandbox, the policy is active and may only allow selected instructions on specific resources. For example, we could limit a processes capabilities to only read files from a certain directory.
In a setting, without additional restrictions, the limits for a process are set by the difference between administrator (root) and a normal user. Meaning, that a process started by the default user is able to access hardware, network resources and files in an nearly unrestricted fashion.
There are multiple scenarios where we may not want a process with such a powerful set of permissions. In a case where this process enters undefined behaviour, meaning in this context that the process executes something that it should not, it may create damage on the targeted system. There are multiple strategies to limited the possible damage and reduce the attack surface. By doing so it is harder for an attacker to achieve his goals, and for an ingenuous user to unintentionally harm the system.
We will introduce different isolation methods and provide an introduction into this topic by looking at different approaches that can be used to isolate processes from each other. We will furthermore conclude by analysing what kind of isolation the Chromium Project (Chrome Browser) uses.

2.2.1 Chroot Jail

`Chroot` is a Linux system call (giving the operating system a command) that allows to create a custom view on the file system for a process. Running `chroot("/home/user2/chrootdir")` will set the 'chrootdir' as new root directory, accessible using '/'. It commonly misunderstood, but the only target is to change the path name resolution. Chroot is not intended do be used for any kind of security purpose or as an sandbox, nevertheless, it is often used as part of a such. It does not restrict filesystem system calls, meaning, it is for example possible to exit a chroot using 'chdir' to change the working directory of the process and then open '../../../etc/passwd'.
Yet, by restricting available file system calls, it is possible to harden a chroot jail [16].

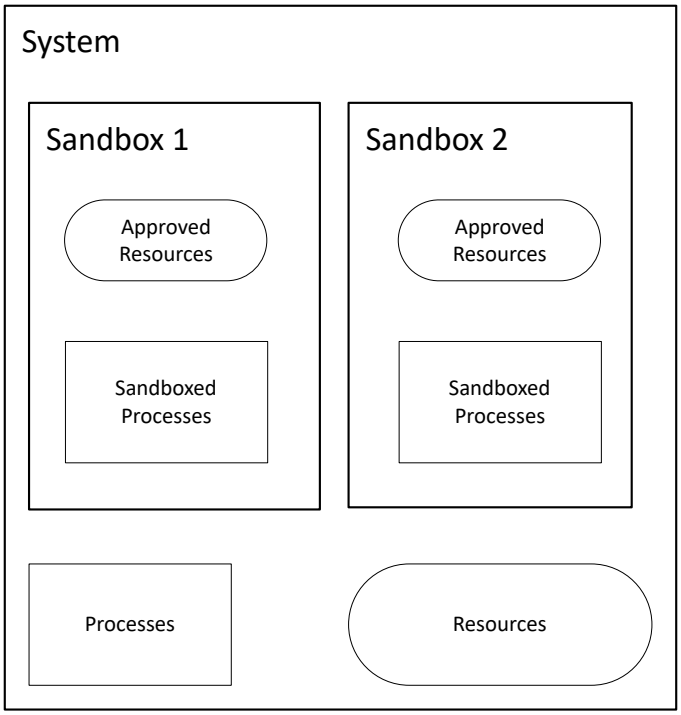


Figure 4: Sandbox Concept

2.2.2 Seccomp(-bpf)

Seccomp restricts an process to 4 possible system calls namely exit, signature, read and write. If a process tries to execute some other calls, it will be terminated. The extended version of Seccomp allows to filter system calls with Berkeley Packet Filter rules. It is possible to use either blacklists, with the downside of a broad diversity of possible system calls, depending on operation system, or as recommended to use a white list as a more secure and stable solution.

2.2.3 Hypervisor based

Figure 5 illustrates the concept of hypervisor based virtualization. There are two types of virtualization, native or hosted. The difference is that the native hypervisor is basically the operating system with direct access to the hardware. The hosted version is on top of an existing operating system installation.
In both cases the hypervisor offers a virtual set of hardware for each virtual machine. The virtual machines, the system that runs on top of the hypervisor, can act like a traditional computer with exclusive hardware, disc and system. This kind of process separation is the strongest type of isolation. It comes with the drawback of additional administrative workload and heavy resource usage.

2.2.4 Container based

Container based isolation is an lightweight alternative to hypervisor-based virtualization. As shown in Picture 6 all instances use the same Kernel and Hardware, provided by the host system. However, the userspace is separated into distinct instances each containing the whole environment for the processes. Each instance, or container, confines the process along with required

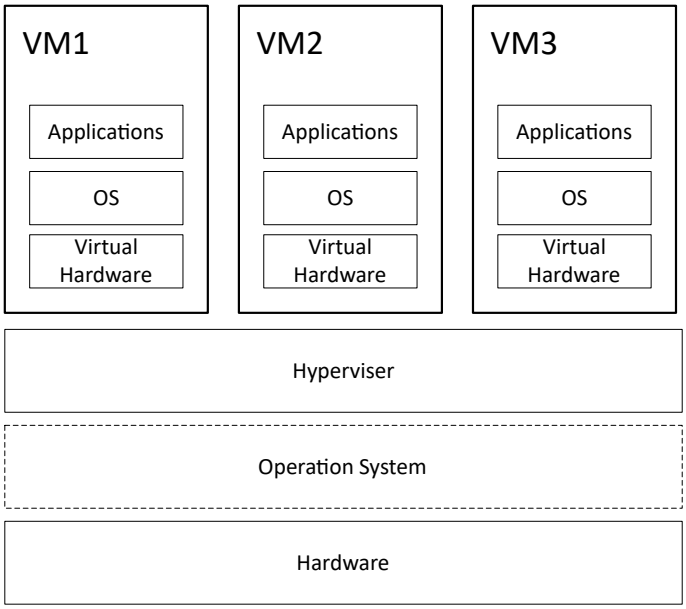


Figure 5: Hypervisor Virtualization

resources, with no access to resources outside of its own container.

2.2.5 Namespaces

There are six different kinds of namespaces available on current linux distributions.
The goal of namespaces is to provide a group or processes with the illusion that they are the only process on the system by isolating specific resources. This can be used for selective security stability measurements and also for lightweight virtualization by creating container, as we read before. The available namespaces are Mount, UTS, IPC, PID, Network and User namespaces [15].
Mount namespaces: Isolates the set of mount points seen by each group of processes. With this namespace environments can be created that are similar to chroot jails, but they are more flexible and more secure.
UTS namespaces: UTS stands for 'Unix Time-sharing System', and is rather old. The namespace to isolate the system identifiers 'nodename' and 'domainname'. Meaning, each NTS can have its own hostname and domainname.
IPC namespaces: Isolates two types of interprocess communication namely System V IPC and POSIX message queues.
PID namespaces: Isolates the PID (program id) number space. Process in different PID namespaces can have the same PID. A very practical benefit from that is that processes can be moved between hosts without changing there PID.
Network namespaces: Isolates all network related components like network devices, IP addresses, IP routing tables. Each container can have its own associated virtual network devices. Therefore, it is for example possible to have multiple containers, each with an service listing on the same port. Routing rules on the host system are used to direct traffic from the separated network to the associated physical network device.
User namespaces: Isolates the user and group ID number spaces. Therefore, a process's IDs can be different inside and outside a user namespace. This creates the possibility to have a process group with an user with root privileges (ID 0) for operation inside of an namespace and unprivi-

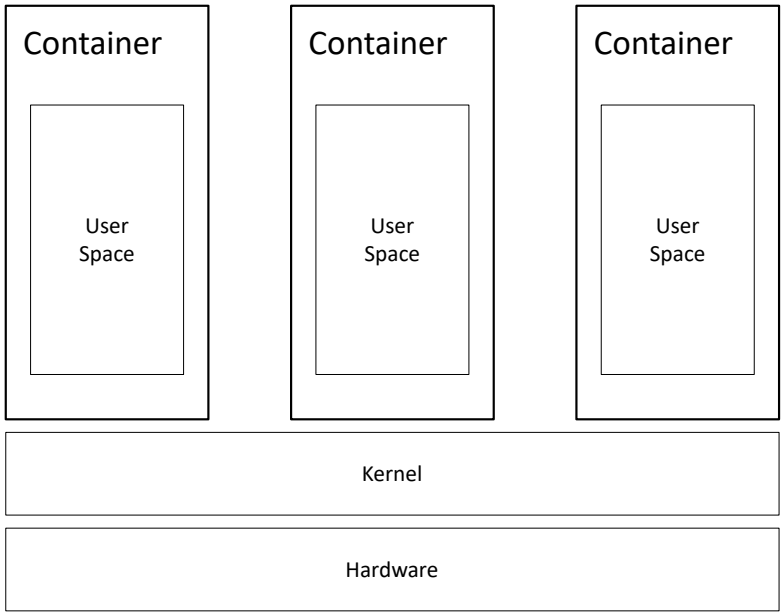


Figure 6: Container based isolation

	Usable for isolation	IPC restriction	User space isolation	Hardware isol.
Chroot				
Namespaces	*	*	*	
Seccomp-bpf		*		
Hypervisor	*		*	*
Container	*		*	

Table 1: Isolation

leged outside of the namespace.

2.2.6 Summary

What we can learn from the above is that there are ready to use, well integrated isolation methods. We can use virtual machines on nearly all hosts or hardware to create a strong separation. By combining chroot and seccomp-bpf we can restrict processes in a way that they can not interfere with other resources on the host system. Furthermore, namespaces allow us to create lightweight and flexible sandboxing for processes, already built into modern Linux systems. The commonly known Docker projects uses namespaces together with other technologies to provide its service [8]. Table 1 provides a short overview over key aspects of the above discussed technologies.
In our implementation, we will try to use a solution that combines namespace and chroot. Our goal is to enable certain processes to work with information that are inserted by the an external sources and limit the possible resources that are available when an attacker finds a way to exploit this process.

2.2.7 Google Chrome

We want to take a quick look onto Google Chrome, to learn how the basic concepts if isolation find its place in a real world software.
While using Google Chrome it creates a sandbox for each tab (Page) the user opens. The goal is to

protect the user from malicious web content. Such web content could be malign code embedded in scripts, pictures, videos and other content that may be integrated into web pages. Furthermore, another goal is to protect the browser as a whole from crashing when a single tab crashes.
The above discussed sandbox mechanisms, beside the hypervisor visualization, were mostly focused on Linux systems. However, Chrome is also available for Windows systems. We will take a look on the Windows solution, in order to see, how sandboxing can be done there. Figure 7 illustrates the Chrome sandbox solution, for Windows.

On Windows each sandbox lives on a separated process. Such system therefore has at least two processes, a privileged controller named broker and sandbox processes often named target. The broker creates the policies (rules) and spawns the target process. Via interprocess calls the broker process receives the actions which the target process wants to execute. If the policy allows the action, the broker executes it and returns the result via the same channel to the target [5].

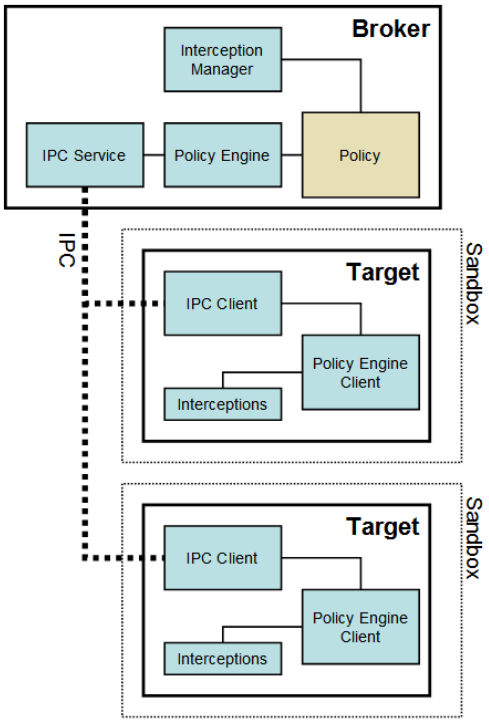


Figure 7: Chrome Windows Sandbox Broker

Source: chromium.googlesource.com

When chrome is executed in an Linux environment, the above introduced methods, mostly namespaces and seccomp-BPF, are used to isolate each tab. The sandbox creates a network and a pid namespace to isolate the process. Furthermore, a seccomp-bpf layer is used, to filter potentially malicious code before execution [4].

3 Rust

We already mentioned in the introduction that the aim of this thesis is to evaluate Rust by creating a personal agent prototype. In this Section we are going to take a closer look at Rust. To do so, a short summary of Rust features that are important in this project, be used to introduce the language. Furthermore, we will pick a few examples and well know vulnerabilities and discuss why they are preventable, using Rust. Overall, this Section should give the reader the insight, why we choose a system language, why this language is Rust and provide the reader with some basic Rust knowledge.

3.1 Why Choose a System Language

We already talked about the effort, that is needed to make such an sensitive topic trustworthy for the public. A security problem is something that would be in conflict with this important requirement. To decrease the chance of design errors in the code, a formal specification can be created and verified by model checker. Based on that specification, test cases can be derived and strict code reviews can be done. For the most sensitive parts of the resulting binary, meaning the generated machine code, should be verifiable. This is easily possible if the generated code is static and readable assembly code. We make this implicitly possible, by choosing a system language.

Furthermore, by removing the underlining framework, the attack surface is reduced. Problems with the .net core or the JVM could introduce new attack vectors. The immense size and complexity of these technologies makes it impossible to provide any guaranty. Therefore, we as the publisher and the clients have to trust these frameworks. Even if we assume that there are fixes for all known security problems before the are exploited, it is still difficult to update all clients. Even more because the concept is that the client is self responsible for its installation. Thinking about the targeted architecture and amount of installations, it will be an easy task to find clients that did not install the latest security fixes. By restricting the critical parts to our own software, we can introduce further restrictions and automated (unpreventable) update systems.

Having in mind that maybe every living person has access to its own personal agent, a small or even embedded solution may be one future step. C is still dominating this area, because of the fine memory control and foreseeable CPU workload. Pushing a .NET installation or a JVM on each of these devices may be an unwanted overhead and comes with higher hardware demands. A program that is directly compilable for common platforms and does not require an additional software seems to be a better choice in respect of the amount of potential users.

In order to further decrease the amount of software systems that the user has to trust, without any change of verification, we may also want to remove the operating system as a whole and run the agent as unikernel application. Using a hardware near language from the beginning, which ideally allows to be run without the standard libraries (operating system depended), makes it more convenient to take this step into directions where the operating system can be removed.

Also, the academic purpose is relevant. Very often system languages are avoided because they are quite complex. Working with such a language, will allow to learn more about the benefits like

speed, transparency and predictability. Also, working on a lower level, increases the understanding for programs and computers in general.

3.2 Rust Features

Rust offers a wide range of built in language features. We will take a look on the most significant features, which underline why we want to evaluate Rust as possible language for the personal agent. Furthermore, we will take a small peak on selected examples to see some Rust code and to underline some important concepts.

3.2.1 Ownership

The initial problem that Rust solves are the memory management problems. In other system languages, it is at least optional it the hands of the developer to decide the moment where memory is allocated and freed. Its commonly known that this leads to many problems like memory leaks, dangling pointers and use after free bugs. Rust solves this issue with a strict ownership model. Each variable has its owner and will be freed as soon as its owner goes out of scope. Often this is the end of the method where the object was created or moved to. Therefore, the developer does not have to care about memory deallocations. The needed operations will be inserted by the compiler during compile time. This allows to use stack and heap variables in a fast and fearless manner, without caring about possible vulnerabilities. The deallocation are compiled into the binary, therefore the overhead for alternative solutions like garbage collection is spared.
Furthermore, the same model allows Rust to guarantee us memory safety. As mentioned already, the Rust compiler frees variables and therefore can verify if we use a value after this free, avoiding use after free bugs. Also, it is not possible for a function, called with a reference, to free this variable and leave the caller with an reference to a invalid memory. Only the owner of an variable, can free it. If the caller allows a called function to take ownership, it will not be able to use the variable any more. This has to be restricted, because of the possibility that this value does not exist any more, after the function call. This is part of Rusts move semantic, where the ownership is moved between scopes. Consider the following example, where we move the ownership of the value 'hello' from the variable x to the variable y. When y goes out of scope (end of function move_var), it will free the value 'hello'. Therefore, the memory where 'hello' was written may be overwritten with another value and accessing this memory area by reading the memory x points to, could lead to unexpected behaviour.

```
1  fn main() {
2   let x : String = "hello".to_string();
3   println!("x={}",x); //this will work
4   move_var(x);
5   //println!("x={}",x); //this will not work
6  }
7
8  fn move_var(y : String){
9   println!("y={}",y); //this will work
10 }
```

Sometimes the move is hidden because the function call is implicit. This is the case when we use

an assignment. In the following example the ownership is also moved:

```
fn main() {
 let x : String = "hello".to_string();
 println!("x={}",x); //this will work
 let y = x; //x will move here
 println!("y={}",y); //this will work
 //println!("x={}",x); //this will not work
}
```

Based on the same concept of ownership, Rust can guarantee threads without data races. Only the unique owner of a variable can modify it by default. Therefore, if an value is used in multiple threads, reading is the only option. If we want multiple threads to modify the same value, we have to use a access control mechanism called Mutex. In difference to some other languages, where a Mutex only locks an area in the program, this Mutex locks one specific variable and allows us to use exactly this locked value.

3.2.2 Strong Typed

Additional to the ownership model, everything in Rust is strong typed and therefore can be type checked. This avoids errors where values are assigned to variables, respectively memory areas, which do not match. The strong typed system also allows type inference This is the reason why in Rust we often see variable declarations with a `let` statement without an explicit type. The following example shows, that there are two ways to create an 32 bit integer. Please note, that even if 1234 would fit into an smaller integer with less bytes, a 32 bit representation is the system default. This type inference can be used everywhere, when the compiler can be derive at some point which type the variable is.

```
fn main() {
 let x : u32 = 1234;
 let y = 1234;
 assert_eq!(x.eq(&y), true);
}
```

3.2.3 Small Footprint

We already talked about the possibility that this agent may run as embedded software on small devices. Or, it might be required to host many instances. The typical small footprint of C makes C attractive for areas where memory and CPU time is rare.
Like C, Rust also has little to no overhead. Even its 'built in' garbage collection does not change that, because it only adds code which should be found in a valid C program. But the Rust compiler adds a little 'runtime' providing a heap, backtraces, unwinding, and stack guards. Compiling Rust without these furthers reduces the binary size, resulting in a very similar binary size that would be the result of a similar C solution.

3.3 Rust is Demanding

Something most people encounter when they start to work with Rust is that Rust is not an easy language to learn. Especially, the ownership and lifetimes concepts are new for many developers.

The community knows this issue and tries to solve these problems with better documentation, videos and tutorials. Nevertheless, the learning curve may be frustrating.
Another challenge is that the Rust compiler. Besides refusing to compile one many occasions, the compiler complaining a lot about bad code style. Naming conventions, wrong set parenthesis, unnecessary mutable or unused variables and unused imports are just a small extract of things that are checked during or before compilation. Especially in the beginning, the Rust compiler will spit out many warnings and tries to motivate the developer to follow the Rust coding conventions. All the warning are often a bit to much at the beginning. For that reason, the can be deactivated, by settling a 'disable warnings' flag. Nevertheless, once the rules are clear and the developer follows these set of rules, particularly future readers benefit from this enforced guidelines.
Altogether, Rust is a very strict language, which is difficult to learn at the beginning, but also comes with numerous benefits to software security. Rust ensures some code equality and quality and allows to write reliable code, independently from the developers experience. This is in great contrast to C/C++ where a lot of experience is needed to write reliable code.

3.4 Cryptography

Cryptography is a complex as well as important part, when we plan the design of our personal agent. We talked about trusting a system during the introduction, encryption is the tool we need to extend the zone, which can be trusted to the surrounding infrastructure. The idea is, that if we can not control the environment, we hide our information from it.
Therefore, we expect support from the existing Rust environment, that allows us to gain quick success for prototypes and further tests. Peaking in the cryptographic area we find that there are already some Rust implementations for commonly known technologies.

- **Ring** is a project that aims to re-implement a core set of cryptographic operations. On top of this implementations, done in multiple languages, a Rust API is provided [27].
- **Webpki** can be used to implement the client side for validating Web PKI (TLS/SSL) certificates. Its uses Ring for signature verification. The Wevpki Team self set goal is to provide the best Web PKI implementation and strictly following the definition [40].
- **Rustls** provides a TLS (1.2 and 1.3.draft) implementation. It uses the the webpki library for certificate verification. Its cryptography is also based on Ring [28].
- **Sodiumoxide** is an easy-to-use Rust wrapper for Salt and Sodium. Sodium is a commonly used C library for encryption, decryption, signatures, password hashing and more [32]. Sidenote: it seems to form a trend that Rust is used to provide an type safe API for C libraries.
- **Rust-Crypto** is a project that aims to implement common cryptographic algorithms in pure Rust. It already provides many hashing algorithms and commonly needed cipher for symmetric and asymmetric cryptography. Some well known provided algorithms are: AES, ECB- CBC- CTR cipher modes, Curve25519, HMAC, MD5, SHA2, SHA3, Whirlpool, ChaCha20 and many more [43].

The research shows, that there are frameworks that provide tools which allow us to implement standard cryptography into our future product. We can secure connections with TLS, sign messages, check signature as well as create asymmetrically and symmetrically encrypted channels.

3.5 Rust vs C

We will now take a look at some common problems in C/c++ that are preventable by Rust. Furthermore, we will talk about the well known Heartbleed bug as example for the weakness that comes with C. We will take a look on what would have happened if OpenSSL would have been written in Rust.

3.5.1 Null Pointer

In C/C++ it is easy to create pointers, which dos not point to a valid address.
Example: `int * pInt = NULL;` .
This is not a problem, as long as the developer is aware of it and handles it properly. Often there are functions that, if everything works as expected return a valid pointer and if something went wrong, they return a Null pointer. A common problem seems to be, that developer forget to handle the exceptional case properly. Handling this and similar cases of seemingly unexpected return values leads to bugs and crashes in programs.
Rust solves this by removing the possibility for a Null value. Instead the enumeration `Option<T>` is used. The Option can be `Some<T>` where T is the expected type of the return value, or `None`, where None represents 'nothing' / null. The idea behind this solution is the following: When the return value is used, Rust can fore the developer to think about both cases by checking if all enumeration values are covered (some, none). Furthermore, when this enumeration is used, it is clear for the developer that this function might return 'None'. A common way is to use a `match` statement in order to handle this. With this statement the compiler forces the developer to take care about all possible cases. Forcing the developer to think a little bit more about the returned value, should eliminate errors with unexpected unset / type-less variables.

3.5.2 Use After Free

In C, it is possible to store the memory address to a variable discretely. Therefore, it is possible to access this address from anywhere in the program. This is even possible, after the memory was freed by the program. What will be written on that particular address is unclear and using it after its freed could lead to undefined behaviour. The ownership model that we already discussed, allows the compiler to check if we try to use a variable or a reference to that variable, after the lifetime of this particular value is over. Therefore, the compiler will refuse to compile the program, if it is not guaranteed that the value is valid.

3.5.3 Race Conditions

In C/C++ we can hand over the reference to an memory address to as many threads as we like. If these threads modify the value the resulting content of this value is depending on the execution order and may not be deterministic any more. Rust solves this with its ownership and secures the value if its used in multiple threads at the same time. The compiler limits the instance that is allowed to change the value, to one instance only. If multiple threads want to modify one single value, a structure has to be used that guarantees that the access is synchronized. This can for example be solved with a Mutex.

3.5.4 Out of Index

In C there are no boundary checks when accessing an indexed type. This means that, as long as no memory is accessed that does not belong to the program, it is possible to access memory before and after that array. Rust does compile to similar binary code but inserts some boundary checks. Therefore, if for some reason a program tries to access memory that does not belong to the given array, the program will panic and refuse to do the access violation.

Overall, these are only a few examples, that should even further introduce Rust as an valuable alternative to C and even C++.

Example - Heartbleed

As mentioned, we want to give an explicit example as finale point in this argument. The question that we want to answer is: would the Heartbleed bug had happened if OpenSSl was written in Rust?
With the information collected possible with the Heartbleed bug an attackers could eavesdrop on communications, steal data directly from services and users as well as to impersonate services and users. It is a bug in a minor SSL/TLS protocol extension called Heartbeat.
Without going into to much details, by sending a HB_REQUEST with a fake payload to an OpenSSL server an attacker was able to use an buffer over-read bug. The amount of bytes that are given in the payload where copied and sent as response. The origin of that problem is that there where no boundary check before the response data are copied to the output. With this over-read it was possible to read the certificate.
If we try to do the same in Rust, the program will crash as soon as we make an attempt to over-read the data buffer. Therefore, the Heartbleed bug, would not have been possible in that damaging extend [10]. Nevertheless, running this without an boundary check, would be an easy target for an DOS attack, even with an Heartbeat extension written in Rust.

3.6 Summary

To somehow get an feeling on how Rust will take its place in this world we can take a look on ongoing projects and companies that already trust this language. First, there is the `rust-lang.org/en-US/friends.html` page. Here we find well known names and how they use Rust. Furthermore, there is an unofficial collection of Rust projects on GitHub, group by there field of use. When looking at `github.com/rust-unofficial/awesome-rust` we see many libraries that get rewritten in Rust, meanly because of the implicit security benefits. Furthermore, we see that C/C++ dominated areas get attention, like cryptography and graphic.
This introduction shows us, that thanks to an ownership model and an strict type system Rust offers an annoying yet helpful and error preventing compiler. Currently it seems to be the best choice to use Rust, with the condition that a system language is required.

4 Specification

In the following section the overall concept is specified. However, the prototype will not necessarily implement the whole extend.
The Personal Agent (PA) is a piece of software. The prototype can be executed on modern operating systems. The goal is to do an identification / authentication off a person for an external entity. The agent can be imagined as a trusted digital briefcase assigned to an unique person, which contains all identities and certificates for different services that belong to that person. A digital identity can be an drivers licence, passport, bankcard, customer card or any other ID or Card that is associated with the user. A certificates can be imagined as tickets, or licenses to access or do certain things, like opening a door, leave a building or enter a train. In the following we will not distinguish between certificates and digital identities, they are both valid responses to an authentication request.

4.1 Definitions

The following will define the most important external parts in this system.

A User is a real living person that uses an personal agent for authentication and authorisation. On the users behalf, a request for authentication will be started by a scanner.

A sensor can be implementer as some sort of biometric sensor, that reads, for example, the fingerprint, the face, the body temperature or any other biometric property of the person. A sensor is connected to one or more defined verifier.

A verifier can be implemented as a device that needs a successful verification to enforce an policy. This could be for example a door to a secured room, or a police officer that wants to know if the person has a license to use a given vehicle class.

4.2 Identification Cycle

The identification cycle is the process that identifies a person and tries to give an authorization for the requested action. Figure 8 shows a basic model for the communication between a already known sensor, personal agent and verifier. The discovery process, where the scanner finds the corresponding agent in the pool of possible agents, is skipped in this illustration. Therefore the scanner, agent and verifier already know each other and are open for new connections within each other.
The process will work as follows:

1. The sensor requests the verification of the person agent.
2. In case the request is valid and the agent has to response, the personal agent sends the verification information to the sensor.
3. After receiving the agent verification, the sensor sends the read data, together with other required informations to the agent.

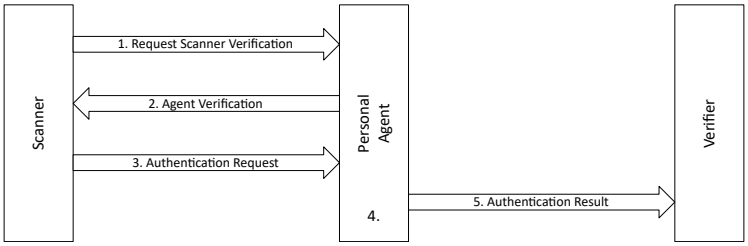


Figure 8: Communication with externals

4. The Agent checks if there is an dataset for the requested service in his storage and if the biometric information matches to one of its templates. This process is part of the Authentication Process which will be discussed in Section 8.
5. In a final step the Agent sends the requested information together with the authentication result to the verifier.

4.3 Code Requirements

Basic expectation on the program and source code style of the personal agent are:

- The agent has to run unsupervised.
- The agent has to be focused on extensibility.
- All source code should be well structured and readable, following basic clean coding guidelines.
- (optional) The agent should be executable on Linux and Windows
- (optional) Follow the idea of least privileges in each module.

4.4 Architecture specification

The following outlines the desired architecture for the prototype implementation.
The communication is based on structural predefined JSON objects. JSON where selected as format to offer a modern and readable structures. The communication contains information that is required for a sufficient demonstration of the authentication process. Sufficient means that it is possible to demonstrate read, write, create and validate operations, on the available content. To create a more relate able experience we tried to follow a very basic self created authentication process. The usefulness of the protocol is limited to this prototype. It should be clear, that this is not a valid or secure protocol for a communication between 3 parties. Nevertheless, based on the demonstration the prototype provides, it should be possible to create a valid protocol, only by adapting the transferred information and used sequences.

4.4.1 Communication with External Entities

Figure 8 already introduced the communication process between scanner, agent and verifier. All the transferred information is wrapped as JSON. Figure 9 and 10 illustrate this JSONs. Later on, we will further define, to which internal data type Rust will map each of these values.

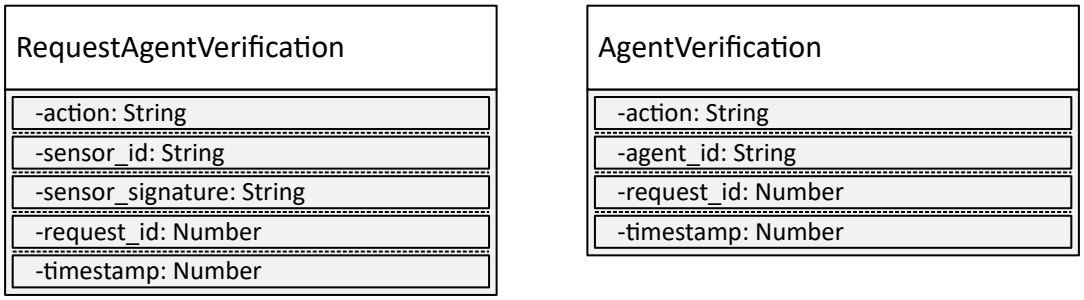


Figure 9: JSON for Agent Verification Request and Response

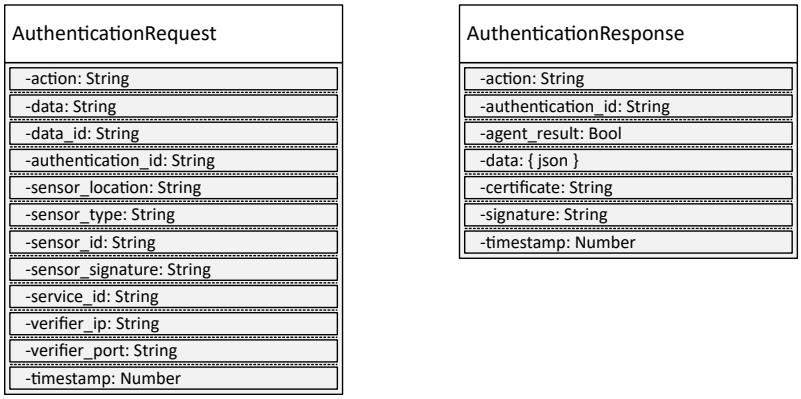


Figure 10: JSON for Authentication Request and Response

4.4.2 Personal Agent Modules

The agent is split into multiple parts - respectively modules as they are called here. Each module provides an well defined service. The modules services are requested by a core module that manages the communication as central point. Each module is separated from the others, so that they should never share the same resources. The communication will be based on TCP based remote procedure call solution.
In the following we will define each of the modules shown in Figure 11 and the core functionality. For each we will define the service and tasks that are required.

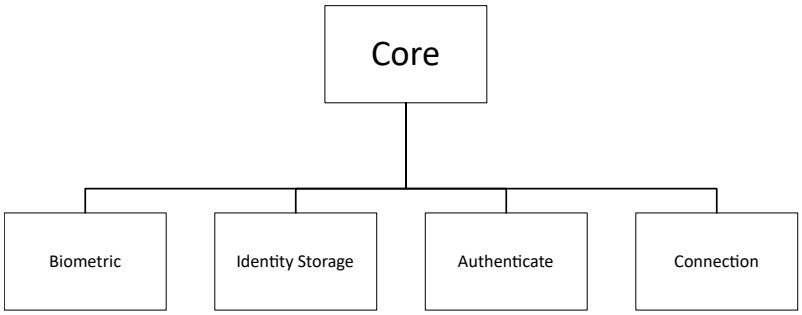


Figure 11: Module Overview

Core

- Holds all information about the effective installation. Each installation has an UUID, a so

called 'Personal Agent ID'. A UUID is used as Agent ID for the transmissions.

- Controls the data flow to and from each module (*Connect*, *Biometric*, *Identity-Storage* and *Authenticate Module*).
- Guarantee that each module only obtain exactly the needed information to fulfil the required task.
- Runs the authentication sequence defined in Section 8.

Connection Module

- The only module that accepts and opens connection to the surrounding environment.
- Opens and closes IP based connections to any given Port, if the receiver allows it.
- Sends arbitrary long data.
- Accepts connections on a given Port.
- Collects incoming messages and allows the core to query them.

Biometric Module

- Has access to biometric templates.
- Is able to compare a given data set with a saved templates.
- Returns the difference between a given data set and the corresponding templates.

Identity-Storage Module

- Has access to the stored identities.
- Allows to query the identity for a requested service.
- Provides an definition for future identities.

Authenticate Module

- Holds the private key for this person.
- Provides a public key.
- Can sign data to authenticate the person.

4.4.3 Assisting Module - JSON Parser and Provider

The JSON module should be accessible for each of the main modules. This and future assisting modules should only be a stateless service provider.
The JSON parser and provider will be used to verify input data and create outgoing data JSONs. A arbitrary string can be handed to the JSON parser, together with an 'expected result template'. The parser will, if possible, return the result as object. The instances provides all required JSON templates / Objects, that are defined in Section 8.

5 Architecture

We will follow the given specification together with the functional description from Section 4 and use it to derive an architecture for the Rust implementation. First, we need to decide where to place which functionality and what interfaces are required. The goal here is to decide on an for an architecture design. Especially, how to split up the different parts of the agent.
We chose that task isolation is the major determining design requirement. The following part will introduce the selected solution and discuss two previously considered possibilities. We will discuss why this solution was selected based on its benefits and drawbacks.

5.1 Task isolation

A modular solution is required, where each module has one distinct field of work. Our goal was to separate these tasks as clear as possible. The idea is, to gain the following advantages:

- No coupling between the separated parts: It should be possible to change or exchange any part without interfering with other parts in the system.
- Independent development of each module: Allowing multiple programmers to work independently on the system.
- Privileges: Tailored privileges for each module.

By separating tasks, interprocess communication becomes an important part of this architecture. To stay independent from operating systems we choose TCP based remote procedure calls, over pips or message queue, as the best solution for the information sharing between each task. We selected TarpC as the most developed RPC solution currently available for Rust [33]. Details about TarpC will be introduced in Section 6.3. For now, let us agree that TarpC is an remote procedure call solution. The key functionality is, it generates all required functions and objects with the help of a Rust macro.
With TCP based RPC, as offered by TarpC, it is theoretically possible to separate processes even on distributed machines, and still share function calls between them [35]. Figure 12 shows the basic concept of such an client-server TCP based RPC solution.

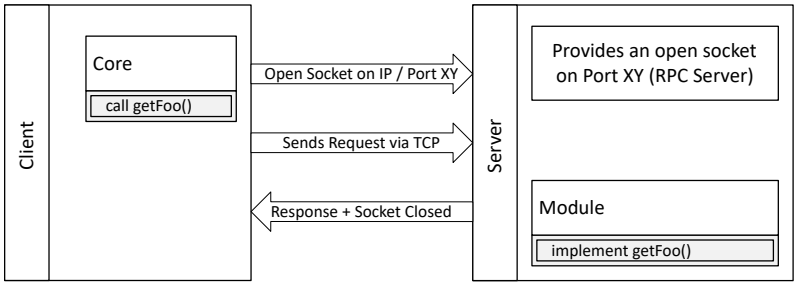


Figure 12: TCP based RPC

We had to find a design that integrates this server and client relationship with our modular design requirement. Altogether, we came up with 3 approaches on how to realize this system:

1. Each module is represented as Rust module. The core can import all needed structures and RPC definitions by bringing the module into scope.

2. Each module is represented as Rust module. The core does not import the macro generated objects. Core and module have to contain the RPC definition and the required object by them self.
3. Each module is its own binary. The RPC definition is contained in a separated library.

In each approach the main difference is the dependency between server and client. The final and third solution that we selected comes with some overhead, but on the other hand allows to strictly separate all tasks. The following in Section 5.1.1 will introduce this solution. Afterwords, we discuss why the other, simpler approaches where sorted out.

5.1.1 Final Architecture Solution

The main concept in our approach is that each part is represented by a separated executable binary. With that, we ensure a high level of isolation between them and allow to make them independent executables from each other. Such a strict separation makes any coupling between the modules impossible.
When we talk about preventing coupling in this context our target is to prevent:

- Objects or memory access from more than one module.
- Any dependencies besides the public interface that would prevent us from changing how the inner solution works.
- Other unintended ways to communicate to the module.

To achieve the third requirement, all communication channels can be limited to the functions that are defined in the RPC interface. Thereby, the all dependencies between modules immediately get clear and structured.
With the selected RPC library, we also introduced a major problem that we had to solve. The library does not provide a common solution to share the interface specifications. TarpC, our selected RPC library, generate all required objects for communication during compile time, exactly where the `!Service` macro is included. We will take a closer look on this macro and the object that this macro generates in Section 6.3. For now, we have to agree on the following: all objects that are required to host a RPC server and to create a client, which is able to connect to this server, are generated where the `!Service` macro is called. Within this macro the RPC interface is defined. Therefore, to be able to use the same interface in multiple binaries, like it is usual in a client / server RPC environment, we have to run this macro in each binary. This will lead to the drawback, that we have to maintain multiple interface definitions, separated into diverse projects.
We solved this issue by extracting that generation code into a separated library. With this library we provide a single point for the interface that will be user for server and client.
As a consequence, each module comes with its own library where the RPC definition is given. This library brings the major benefit that we provide a single point to define our interface and look up the function headers. Figure 13 illustrates that concept.
In the resulting design each part is its own application and can be developed separately without any risk of unintentionally changing the defined interface. We see this as beneficial asset in a project, where each module will need a lot of attention in order to get it ready to use in a real world scenario.
Moreover, the separated binaries also enables us to give each part unique privileges by putting

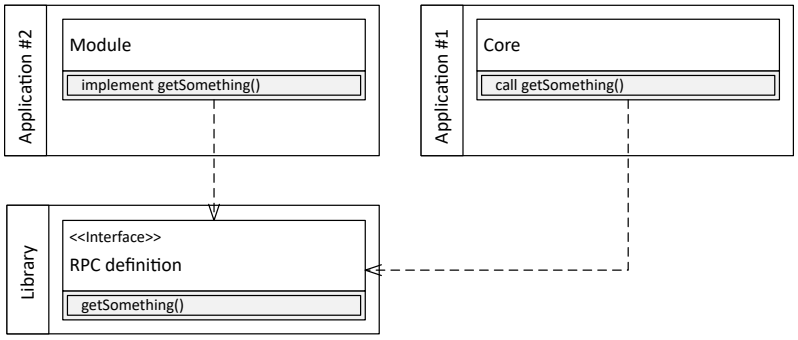


Figure 13: RPC Design for Module Architecture

them into sandboxes which we already introduced in Section 2.2. Therefore, in a future step it will be possible to only allow a module to read and write in required directories and only allow to communicate on the ports that are required for the RPC communication. With that, we can potentially increase the security for scenarios where single modules gets infected.

Another aspect we have to discuss a drawback that comes with this solution. The pure socket based communication between the parts may be heavily used for each request. Therefore, a lot of serializing and deserializing is done for basic function calls that would, without RPC, require only some stack operations. In fact, we increase the duration of each request with the RPC overhead.

Nevertheless, summing up the benefits and drawbacks shows us that this solution is a satisfying way to go, based on the considered aspects.

+ Separation of tasks
+ Information hiding for each part
+ Direct coupling is not possible
+ Single point interface definition
+ Very easy to extend and maintain each module
- Complex, many separated tasks needed
- Compared to non TCP based RPC solutions, slower and traffic intensive

On the one hand we introduce a good way to provide task separation. On the other hand this scheme introduces a comparatively high complexity for the currently planned functionality of the prototype. However, in a future implementation of this solution, where this might be the cornerstone that results in a good overview. Also, a possible formal verification would be simpler when tasks are clearly separated. Moreover, logical bugs can be found easily when its clear that a given source code only has to care about a given set of operations, files and data. Therefore, the remaining drawback is the throughput.

Core Module Pull vs Push We decided that the core does not implement an RPC server and acts only as a rpc client. Therefore, if the core module need data or wants a module to perform a certain task it will pull the data from another module. In fact, the decision is most relevant for the

interface between the connection module, which receives requests from the outside, and the core, the module that has to process the requests from the outside.
A naive approach would be that the connection module pushes the received data, respectively a authentication request, to the core module which than processes this request. On the other hand, the core module can implement an mechanism to pull incoming authentication request iteratively from the connection module.
The decision was made based on the assumption that the agent has to perform complex tasks in order to verify and authenticate the user, in a future version. Executing this assumed tasks may take several seconds to complete. With this constraint in mind, we choose the pull variant. The core module can decide the frequency or event it wants to query a new request to process. With that the core can protect itself and the modules it uses from potential overloading by to many request. Furthermore, simply by setting a fixed low enough pull frequency, the core module can create real time promises for the response generation.

The critic here is that if each incoming request has to be pulled by the core and then sent to various modules over RPC, it could potentially make the agent to slow by design. The worry might be that the core will get to slow to generate a response for an authentication request within an acceptable time frame. We will take a look onto that in Section 11.2, by testing this assumption with our prototype.
The final architecture is illustrated in Figure 14.

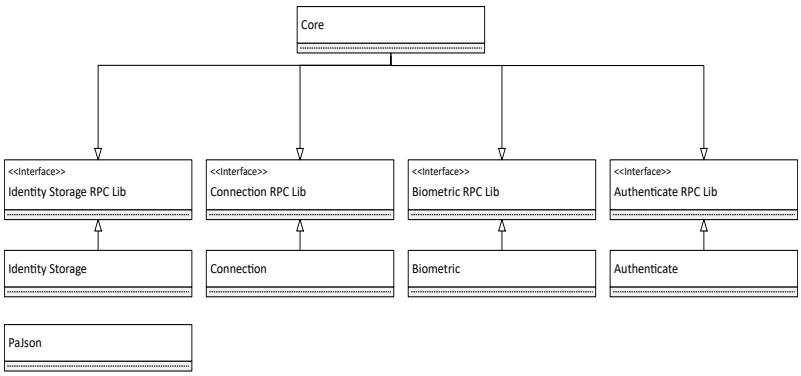


Figure 14: Module and RPC Design Integration

We had several other architecture prototypes that where used to collect experience and where improved multiple times, until the design resulted in this introduced solution. The most distinguishable steps are introduced in the following. Nevertheless, they where not satisfying because of several severe drawbacks, which we will also discuss.

5.1.2 First Approach - Use Rust Modules

The first approach followed the examples given on the TarpC GitHub page. This solution suggests that one single application implements both, server and client.
Because our module design is crucial and Rust offers modules, it was a logic step to separate the functionality intro these Rust modules. Therefore, the main program acts as client and a set of Rust modules act as RPC Server.
The required objects for the RPC communication, are generated in these modules. With that the

core can easily work with the generated objects by importing the modules into the scope of the core.
But because Rust modules are just a logical separation to structure code, bringing them into scope with `use modulename;` also comes with a major drawback. We break our focus on separation by introducing some direct bounds between core and module. This step makes many unwanted things to easy. A developer will always be tempted to use shared objects contained in these modules and circumvent the communication via RPC.

Additionally, we think that using RPC like that, does not make much sense. Rust offers us better tools like channels, that could be used for this kind of communication, where the processing is done in multiple threads with the same origin (Main Program).
Summing up the pro and cons leads us to:

+ Easy to use: We have direct access to all fixed and generated objects we need
+ All in one: We only have one build.
+ All in one: Its easy to keep an overview.
- Coupling: We are forced to access objects and share memory.
- All in one: Developers will be tempted to create more global, connected objects.
- All in one: Imagine a project with more then one developer, this will make task separation harder.
- Channels: There are better solutions than RPC communication to implement that.

Because of the easy and unpreventable coupling this solution was rejected.

5.1.3 Second Approach - Using RPC in Rust-Modules

The second solution is similar to the first one, but with a focus on the basic idea of using RPC for separation, like its shown in Figure 12. This approach still builds on the Rusts module system and work with one binary, but we limited the access to the modules to the RPC calls from the core. It is still just a self set rule that can easily be broken. On the other side, it is also easy to verify, if the developer followed the guideline.
Making this separation this strict introduces another unwanted drawback. Our RPC framework needs objects that are generated with the `service!` macro for both sides, server and client. In order to have these object available, we needed to write the service definition twice, once in the module and once for the core.

```
1        service! {
2                rpc hello(name: String) -> String;
3        }
```

It is required to maintain the definition on both side, client and server. Otherwise, the client is not able to generated calls for the RPC interface.
If the definition is out of sync, runtime errors will occur. Defining this twice makes it impossible for the compiler to do his strict type check for both sides. He does not know that we only generate the definition so that the type checker, when checking the RPC function call, knows what the return value should be. The type checker does not know, that we will call a remote function

that might return another object. If the return value does not match, the deserialization will most likely fail.
Furthermore, another drawback is that we have RPC definitions in our client that are not implemented. This might be irritating and also might be considered as bad style.
Summing up the pro and cons leads us to:

+ All in one: We only have to maintain one binary
+ RPC: Using RPC without any coupling.
- Maintain: Need to maintain the RPC interface definition twice.
- Error: Possibility for runtime errors
- RPC in Core (Client): We may have to add dummy methods to the core.
- Coupling: It is still easy to introduce direct connections

Because of the uncomfortable drawbacks with the RPC client, maintaining and possible runtime errors, we had to evolve this idea to the already presented final solution, where we centralized the RPC definition into own libraries.

6 Frameworks

During the last section, we discussed the architecture and mentioned already one of the used library, namely TarpC. In this section, we want to further outline the functionalities of this TarpC library as well as other selected libraries. The goal is to introduce the indented usage of these frameworks.

6.1 Gaol

Gaol provides an application sandbox, written in Rust. We want to us Gaol to create a layer of defence against malicious or accidental operations. With Gaol it is possible to create an operation - white list and spawn restricted child threads that can only use certain operation that are including the white listed ones [44].
The example below shows an Gaol implementation, and is annotated with explaining comments. We removed statements that are irrelevant for the understanding and focused on the essential parts.

```
extern crate gaol;

// Create the sandbox profile.
fn profile() -> Profile {
 // Set up the list of white list operations.
 // This is what the child thread is able to perform.
 let mut operations = vec![
  Operation::FileReadAll(PathPattern::Subpath(PathBuf::from("/lib"))),
  Operation::FileReadAll(PathPattern::Literal(PathBuf::from("/etc"))),
  Operation::NetworkOutbound(AddressPattern::All),
  Operation::SystemInfoRead,
 ];
 Profile::new(operations).unwrap()
}

fn main() {
 match env::args().skip(1).next() {
  Some(ref arg) if arg == "child" => {
   // This is the child process.
   ChildSandbox::new(profile()).activate().unwrap();
  }
  _ => {
   // This is the parent process.
   let mut command = Command::me().unwrap();
   Sandbox::new(profile()).start(command.arg("child"))
 ↪  .unwrap().wait().unwrap();
   // Will start the same binary again with the child argument set
 }}}
```

The sandbox uses a profile in which we can define the granted operations for the child thread. We can allow access to a given with `::Subpath`. In the example the thread can access all files that are in
lib and its sub folders, recursively. Also, it is possible to allow only the access to one specific folder with `::Literal`. Moreover, we can allow the thread to read the current System Informations

and we can specify which outgoing connections are allowed. What is not in the example, we can also restrict the access to only the meta data of files and directories.
In the example we already see the rather uncommon way to start the application. At first, we check if the argument 'child' is set, if not, we assume that it is running a main (starting) thread. In the main thread, we create the profile and spawn a new instance of the same program, but now started with the 'child' attribute set. When the attribute is set, the process is detected as child process and we start into the first branch, immediately activating our `ChildSandbox`. After that, the restrictions are active.
Our target is to use Gaol to restrict our modules to folders that they need.
When we take a deeper look under on Gaol we find some of the technologies that we already discussed in Section 2.2 about isolation. On a modern Linux kernel, namespaces and chroot is used to create a sandbox. Diving into namespace.rs we see that Gaol does the following:

1. Create a chroot for the 'FileReadAll' policies.
2. Drop capabilities for the mounted directories.
3. Prepare the IPC, mount, UTS, User and PID namespace.
4. If restricted (not in the profile), a network namespace is added to the preparation.
5. Using `Unshare` [18] to change the execution context (activate most namespaces).
6. Starting new child thread in new namespace (active pid namespace).

There are also libraries that use only seccomp-bpf to restrict the network traffic, but we could not find the link where this is used. Furthermore, the developer comments that this is the weaker approach because it is limited and it might not be used.

6.2 Futures

A future, sometimes called a promise, is a representation of an single asynchronous computation. This computation may result in an final value or an error. The most prominent consumer of Features in the Rust world is Tokio.rs, a platform to write networking code. Also the following framework on which we take a look on, TarpC, provides a mode in which the communication is done asynchronous with futures.
A Future is basically only a trait that can be implemented.

```
pub trait Future {
 type Item;
 type Error;
 fn poll(
  &mut self,
  cx: &mut Context
 ) -> Result<Async<Self::Item>, Self::Error>;
}
```

In the code snippet we see for the item that should be returned, any type can be used. When we call poll the result could be the expected value, `Ok(Async::Pending)` or an Error. Most certainly, we will not call the poll method directly.
The future module that contains the trait, also provides multiple options to handle futures and their results. For example the `AndThen` chains another Request to an Future, when the first one

has finished and returned with an `Ok`. We can also use `MapErr` to handle errors or a simple `Then` that will run, independent of the previous result.
Typical use cases for Futures are database queries, RPC invocations, long running CPU-intensive task and reading bytes from a socket. We will not implement our own Features by hand but we will use generated futures [36]. The following example demonstrates a simple use case:

```
fn main() {
 let addr = "127.0.0.1:6142".parse().unwrap();
 let listener = TcpListener::bind(&addr).unwrap();
 let server = listener.incoming().for_each(|socket| {
  println!("accepted socket; addr={:?}", socket.peer_addr().unwrap());
  Ok(())
 })
 .map_err(|err| {
  println!("accept error = {:?}", err);
 });
}
```

In the example an open socket is listening for new incoming connections on Port 6142.
`listener.incoming()` provides us with incoming structures. The incoming structure implements a trait called Stream. Streams implement the Future trait and we can use the introduced asynchronous executed functions. Furthermore, the incoming object contains a TCP stream and a socket address. We use `for_each` on this stream to handle each socket. `For_each`, is an custom extension to the basic Rust Futures. Moreover, we use the already introduced `map_err` to print out errors, if they occur.

6.3 TarpC

TarpC is a project that aims to make Remote Procedure Calls in Rust as easy as possible. We already saw that it actually takes a lot of work away from the developer. This is a huge benefit, but also comes with some requirements in understanding how it works. Especially, when something is broken.
TarpC offers us two modes of communication. A synchronous, where the calling thread is blocked, and an asynchronous where we use futures and do not block the main thread. The prototype uses only asynchronous calls, therefore, we will focus on this type of calls.
TarpC is uses both Futures and Tokio, in the background. The repeating work for each TarpC instance is done within the `service!` macro. This macro defines the interface. The macro will create the client and the server object for us. All we have to do, is using the server object to start the RPC service. On the other side, the client object is used to connect to the server. After that, we are able to call the defined function. The response will implement be wrapped in a Future, that contains an error when something went wrong, or the requested result with the defined type [33]. We will see all the described steps in the following example.
The `service!` macro will create a server and a client. After implementing the functions for the generated function headers on the server side, the client will be able to call the `fn hello` function.

```
service! {
 rpc hello(name: String) -> String;
```

```
}
```

As mentioned already, the macro alone is not enough. We have to implement our service on the server side. A common pitfall is to miss, that the return type for the RPC function, has to be a combination of the function name `hello` and `Fut` for Future, for example `HelloFut`. The fact that the return type is different then in our specification, may confuse at first sight. We will return a Future that encapsulates a Result which in turn wraps our defined return type. A Result is a special Rust enum that can wrap around an object that implements the error trait (Err) or an arbitrary object (Ok). Results are used in Rust to avoid return values like 'null' or similar. Because of the result type, the function has to return with `Ok(string_value)` or `Err()`
The following is the implementation of the hello function that we defined in the Rust code above.

```
#[derive(Clone)]
struct HelloServer;
impl FutureService for HelloServer {
 type HelloFut = Result<String, Never>;

 fn hello(&self, name: String) -> Self::HelloFut {
  Ok(format!("Hello, {}!", name))
 }
}
```

After implementing the function, we still need to start the RPC server and connect the client.
The first step is to start the server. For that, we use the HelloServer that we implemented in the above code snippet. We start the server on the first free socket address and use the returned handle, to hand over the information to our client. A step that is easily missed is to actually spawn the server in the reactor. The reactor is part of tokio. It is a simple event loop. In the final step, we are now able to use a rector to run our future. If no error ocures, we continue with the `and_then` part where the client that was returned from the connection, is used to call `fn hello`. Finally, we map the future to an printable `resp` variable and print that.
The important part that TarpC did for us was implementing HelloServer and the FutureClient.

```
fn main() {
 let mut reactor = reactor::Core::new().unwrap();
 //Server
 let (handle, server) =
 ↪  HelloServer.listen("localhost:10000".first_socket_addr(),
  &reactor.handle(),
  server::Options::default()).unwrap();
 reactor.handle().spawn(server);
 //Client
 let options = client::Options::default().handle(reactor.handle());
 reactor.run(FutureClient::connect(handle.addr(), options)
  .map_err(tarpc::Error::from)
  .and_then(|client| client.hello("Mom".to_string()))
  .map(|resp| println!("{}", resp)))
  .unwrap();
}
```

We saw a introduction into TarpC and we will use this basic knowledge about TarpC again when we look at how we implemented it into our personal agent.

6.4 Serde JSON

JSON stands for JavaScript Object Notation and is a human readable open-standard format. A JSON objects consists of a collection of key value pairs.

```
1  {
2   "matnr" : "k12345678",
3   "age" : 20,
4   "grades" : {
5    "secureCode" : 1,
6    "software" : 2,
7   },
8   "open" : ["math1",15614],
9   "active" : true,
10  "failed" : null
11 }
```

The above example shows an typical JSON and contains all valid types. Following types are possible: Number (Line 3,5,6), String (Line 2), Boolean (Line 9), Nullvalue (Line 10), Array (Line 8) and Objects (Line 4).
We will use JSONs as communication format with external parties like a scanner or verifier. Therefore, a reliable JSON reader, validator and creator is needed.
Serde JSON offers what we need to meet this requirements. **Serde** is one important part in that framework [29]. Serde is a framework for serializing and deserializing Rust data structures. In order for Serge JSON to work properly, we need to make our Rust objects serializeable. In case we do not want to implement that by our self, serge offers us to implement that for us. All we have to do is writing `#[derive(Serialize, Deserialize)]` above our Rust object that represents the JSON. Serde JSON is able to create a JSON from text for us and vice versa. When parsing a string that represents a JSON we have two essential modes of operation [45]. The first mode, shown in the following code sample, will parse any JSON and will provide us with a collection of JSON Values.

```
1 enum Value {
2  Null,
3  Bool(bool),
4  Number(Number),
5  String(String),
6  Array(Vec<Value>),
7  Object(Map<String, Value>),
8 }
```

The following example shows, how we can parse any string representation into a serde_json object that consists of the above values.

```
1 fn untyped_example() -> Result<(), Error> {
2 // Some JSON input data as a &str. Maybe this comes from the user.
3 let data = r#"{
```

```
 "name": "Peter Parker",
 "age": 30,
 }"#;
// Parse the string of data into serde_json::Value.
let v: Value = serde_json::from_str(data)?; //? try or return with
 ↪  Error
// Access parts of the data by indexing with square brackets.
println!("Name {} age {}", v["name"], v["age"]);
Ok(())
}
```

The second mode will try to parse a JSON into a valid Rust object if possible.

```
#[derive(Serialize, Deserialize)]
struct Person {
 name: String,
 age: u8,
}

fn typed_example() -> Result<(), Error> {
 // Some JSON input data as a &str. Maybe this comes from the user.
 let data = r#"{
  "name": "Peter Parker",
  "age": 15,
 }"#;
 // Now we are asking for a Person as output.
 let p: Person = serde_json::from_str(data)?;
 // Can be accessed like any other Rust data structure.
 println!("Name {} age {}", p.name, p.age);
 Ok(())
}
```

Thanks to this mode, the JSON parser ensures for us that all values match directly to what we expect, including number rangers. If any value does not match to the defined type, the parser will return with an error.
The third function we will need for the agent implementation is to create a serialized JSON out of a given Rust structure. Again the structure has to derive from Serialize, and Deserialize.

```
#[derive(Serialize, Deserialize)]
struct Address {
 street: String,
 city: String,
}
fn print_an_address() -> Result<(), Error> {
 // Some data structure.
 let address = Address {
  street: "Prager Straße 15".to_owned(),
  city: "Linz".to_owned(),
 };
 // Serialize it to a JSON string.
 let j = serde_json::to_string(&address)?;
 // Print, write to a file, or send to an HTTP server.
 println!("{}", j);
 Ok(())
```

```
  }
}
```

Altogether, Serde JSON is a helpful library that can increase the security be checking the type of untrusted data and providing an easy way to serialize a given structure.

6.5 Rust-Crypto

In order to use TLS on our RPC communication we need some cryptography. Rust-Crypto aims to collect common cryptographic algorithms and tries to offer a 'as far as possible' Rust implementation for them. However, some solutions still need some assembly. Furthermore, the aim to create an auditable solution, but they are currently not thoroughly audited. Therefore, they still appeal to not use this library if cryptographic security is important. Therefore, before the agent goes live the used algorithms should be audited or another solution has to be found.
We will use PKCS12 to wrap our X.509 certificate, which we will use to encrypt the communication between processes. Later on, we will take a lock how this is used in our implementation [6].

6.6 UUID

A UUID is unique 128-bit number, stored as 16 octets. We will use UUID to trace requests and identify entities in our system. Version 4 UUID is a random number and will be used for this implementation [37].
An example for such an UUID is '1e3b15cd-d9a6-4f3c-adbc-61de9df21401'. With the UUID framework we will parse and if needed create own UUIDs. The following example, shows how this could be done:

```
fn main() {
 let my_uuid =
 ↪  Uuid::parse_str("1e3b15cdd9a64f3cadbc61de9df21401").unwrap();
 println!("Parsed a version {} UUID.", my_uuid.get_version_num());
 println!("{}", my_uuid);

 let my_uuid = Uuid::new_v4();
 println!("{}", my_uuid);
}

/*
Parsed a version 4 UUID.
1e3b15cd-d9a6-4f3c-adbc-61de9df21401
b27c48ae-454b-4cd1-af23-375c2da823c9
*/
```

If we want to create UUIDs we have to activate that feature in our dependencies by changing the standard uuid reference to:

```
[dependencies]
uuid = { version = "0.6", features = ["v4"] }
```

6.7 Lazy Static

Lazy static is a macro that can be used to declare a lazily evaluated static. A regular Rust static has to be fixed at compile time. Moreover, it is not possible to create an static value that requires a heap allocation like vectors and hash maps. The rule is that function calls in statics are limited to struct and enum constructors.
The macro enables us to create dynamic static objects that are evaluated on their first call.
The lazy static also hides the insecurity of mutable static variables from us. A regular mutable static has to be modified in an `unsafe` block, as shown in the example:

```
static mut N: i32 = 5;
unsafe {
 N += 1;
 println!("N: {}", N);
}
```

To generate some degree of secure access from multiple threads the sync trait is implemented for lazy statics. This implies each type we want to use for a lazy static also has to implement this trait. Some example for lazy statics are shown in the following example:

```
lazy_static! {
 static ref HASHMAP: HashMap<u32, &'static str> = {
 let mut m = HashMap::new();
  m.insert(0, "foo");
  m
 };
 static ref NUMBER: u32 = times_two(21);
}
fn times_two(n: u32) -> u32 { n * 2 }
```

What we see is that we can have static hashmaps that are generated on first call. Its also possible, to call arbitrary functions when creating a new static variable. When we want to use a lazy static, we have to dereference it with the `*` prefix
For example: `println!("Results: {}.", *NUMBER);`

7 Frameworks Usage

By following the specification introduced in Section 4 we derived and introduced the targeted architecture in Section 5. We also introduced the tools (frameworks) we want use to accomplish our goal in Section 6. In this section we will discuss how we used the previously introduced frameworks.

7.1 TarpC

TarpC was one of the first technology decisions we made. Therefore the RPC framework it influenced many design decision that lead to the current result.
We already mentioned while introducing the final third solution for module separations that we have separated libraries in which the interfaces are defined. The following example shows such an interface definition. This is the definition for our identity storage.

```
pub use FutureService as IdentityService;

service! {
 rpc get(service_id : String) -> Option<String>;
}
```

We have to implement the IdentityService as public, because this is the object that has to be shared between the caller and callee. The IdentityService later provides informations about the implemented function signature for the caller and the required function that the callee has to implement.
The following example shows such an implementation, again taken from the identity storage.

```
extern crate identity_storage_rpc_lib as rpc;

#[derive(Clone)]
struct IdentityServer;
 impl IdentityService for IdentityServer {
 type GetFut = Result<Option<String>, Never>;

 fn get(&self, service_id: String) -> Self::GetFut {
  Ok(handle_get(service_id))<<
 }
}
```

We see that we implement the previous discussed public IdentityService for a local IdentityServer structure. We define the Future that we will return for an incoming request as the 'GetFut'. Again, please note that we can not vary that name.
The RPC caller also has to import the library with the definition. After the import it can use the public object and the known IP and Port to connect with the RPC server.

```
reactor.run(identity_storage_rpc_lib::FutureClient::connect(SocketAddr
 ↪  ... ))
 .map_err(tarpc::Error::from)
 .and_then(|client| client.get(service_id))
 .map(|resp| { ...
```

We will get an client object as result of the connect call. This object contains information about the provided functions and there return type. We can use future functions like `map_err, and_then, map` to handle this request and incoming results.
For the prototype, we found a flexible and transparent solution for how to work with TarpC. With a view on scale ups the solution is quite robust and expendable in terms of complexity.
Nevertheless, there where serious problems at the beginning to understand the functionality of TarpC. From these problems, we can derive that TarpC is not the best starting point to get to know Rust.

7.2 TarpC with TLS

When someone inspects the traffic between the core and its modules he will find all RPC messages in clear text. The communication is based on simple TCP messages. TarpC has optional TLS support to secure this communication. Thanks to that, we are able to encrypt all the communication between the core and its modules.
By attaching a so called TLS acceptor to our RPC server, we enable the endpoint to create such an connection.

```
//Server
fn get_acceptor() -> TlsAcceptor {
 let buf = include_bytes!("../../../certs/certificate.p12");
 let pkcs12 = Pkcs12::from_der(buf, "password").unwrap();
 TlsAcceptor::builder(pkcs12).unwrap().build().unwrap()
}
...
AuthServer.listen(address, &reactor.handle(),
 ↪  server::Options::default().tls(acceptor)).unwrap();
//Client
let options = client::Options::default()
 .handle(reactor.handle())
 .tls(tls::client::Context::new("core.personalagent.at")
 .unwrap());
```

The above example shows the two changes that are needed on the server side. To create the TlsAcceptor we created a function, that reads the certificate during compilation and with that integrated it in the binary. Therefore, we would have to update the modules before the certificate expires. We see this as an security feature, where some level of forced update control can be applied. Imagine a real world scenario, where a user may forget to update the resulting software, they will no longer be able to communicate with the modules when updates are not installed.
Currently the connections between core and modules can only be monitored on the loopback interface, because in the prototype everything runs on one host, this is not a common security problem as long as the host is reasonably secured and hosts only one agent. In a future scenario, this may be distributed over multiple nodes where network sniffing is possible. This is the most common case, where TLS is required
To be able to use this a valid and trusted X.509 certificate is required. It turns out to be a common issue to create a self signed testing certificate which is suitable in this domain. To help and archive that process for the future, the following console example shows how we generated our .p12 certificate.

```
openssl genrsa -des3 -out server.key 1024
openssl req -new -key server.key -out server.csr
  Country Name (2 letter code) [AU]:AT
  State or Province Name (full name) [Some-State]:Austria
  Locality Name (eg, city) []:Linz
  Organization Name (eg, company) [Internet Widgits Pty Ltd]:Personal Agent
  Organizational Unit Name (eg, section) []:Core
  Common Name (e.g. server FQDN or YOUR name) []:core.personalagent.at
  Email Address []:n@e.mail
cp server.key server.key.org
openssl rsa -in server.key.org -out server.key
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
cp server.crt server.crt.pem
openssl pkcs12 -export -out server.p12 -inkey server.key -in server.crt.pem
```

The critical information that is needed for the TarpC configuration are the password and 'Common Name' attribute. Furthermore, it is required to add this certificate to the 'Trusted Root Certification Authorities - Certificates'.
After a successful certificate installation, the communication can be secured. We will take a verify if this actually does what we expect, by inspecting the traffic with and without TLS in Section 10.2.5.

7.3 Data Validation with SerdeJSON

We discussed already some details about the `pa_json` library. Furthermore, we want to underline how the combination of a strong typed Rust JSON parser and our provided `trait UntrustedJson` adds another layer of security.
Deserialize with SerdeJSON allows us to use bool, i8, i16, i32, i64, u8, u16, u32, u64, f32, f64, char and str as type constrains. Trying to parse a u16 value for a JSON value that we defined as u8 will lead to an empty result [45].
On the other side, we also make use of the generic parsing, when we parse an identity from the drive in the identity storage. Here we use the `serdeJson::value` field for the variable content of an identity and only enforce some parts in this stored identity.

```
pub struct Ident{
 pub content : Value,
 pub sha512 : String,
 pub rsa_public : String
}
```

The example shows, that we can combine the fixed type constrains (the two Strings) and add variable fields that has to be given but its content can be whatever is required.
It is important to ensure a secure parsing from objects that with an outside source. We can not trust any of the external sources. Incoming JSON with request to the personal agent will be one of the first targets during a search for security leaks. Therefore, we mark this JSON with a special trait named `UntrustedJson`. This trait provides a method named `fn check` that forces the developer to think about further restrictions that are not covered by simple type checks.

```
impl UntrustedJson for RequestAgentVerification{
 fn check(&self) -> bool {
  Uuid::parse_str(&self.sensor_id).is_ok()
 }
}
```

For example, we use this to ensure a valid UUID for incoming requests.
This combination enables us to implement a strict interface.

7.4 Sandbox with Gaol

The goal is to demonstrate and test Gaol as an possible sandbox solution. We managed to restrict the part of the identity storage that reads from the folder that contains the identities, to be only able to read in this folder without any exception. The challenge was to combine the RPC server in the module with the creation and use of the sandbox, as Goal is intended to be used.

After some problems that we will reflect later, we derived the following process. The approach we us is similar to the architecture that is used for Google Chrome. The main identity storage thread, will act as the broker process, that spawns the child thread if it is needed. As soon as an RPC call arrives, we open an TCP Socket. This TCP Socked will be used by the future child thread to transmit its results back to the main thread. After the TCP Socked was opened by the main thread, a child thread is spawned. The child thread is spawned by calling the identity storage with a special set of parameters which contain all needed information. This information include the file we want to read from, together with a parameter that tells the idenitiy storage application, that we want to start the child thread, instead of the usual main thread. After successful spawning the child, it will read the file. If the read operation was successful, it will send the content to the TCP listener that will afterwords close itself. Now the main thread can return the result to the RPC caller. Figure 15 illustrates the internal process of the identity storage.
By using this approach we can gain some benefits: Neither the core thread nor the identity storage main thread have to investigate the given path. Simply by restricting the part of the program that reads the file, we can ensure that, if a read was possible, it was a valid operation. Furthermore, such a child thread can be used to handle sensible data, without any change of exposure.

7.5 Problems with Gaol

In Section 7.4 we introduced our solution how a sandbox can be used in combination with our TCP based RPC solution TarpC 6.3.
During the Gaol integration, we encountered multiple difficulties that will be discuss here. The key aspects that occupied us where the following:

- We could not connect to the RPC listener within the sandbox.
- The sandboxed child thread was not killed with the parent thread.

This discussion should further underline why we choose the solution we implemented. In Section 6.1 we introduced Gaol, and showed the intended usage.

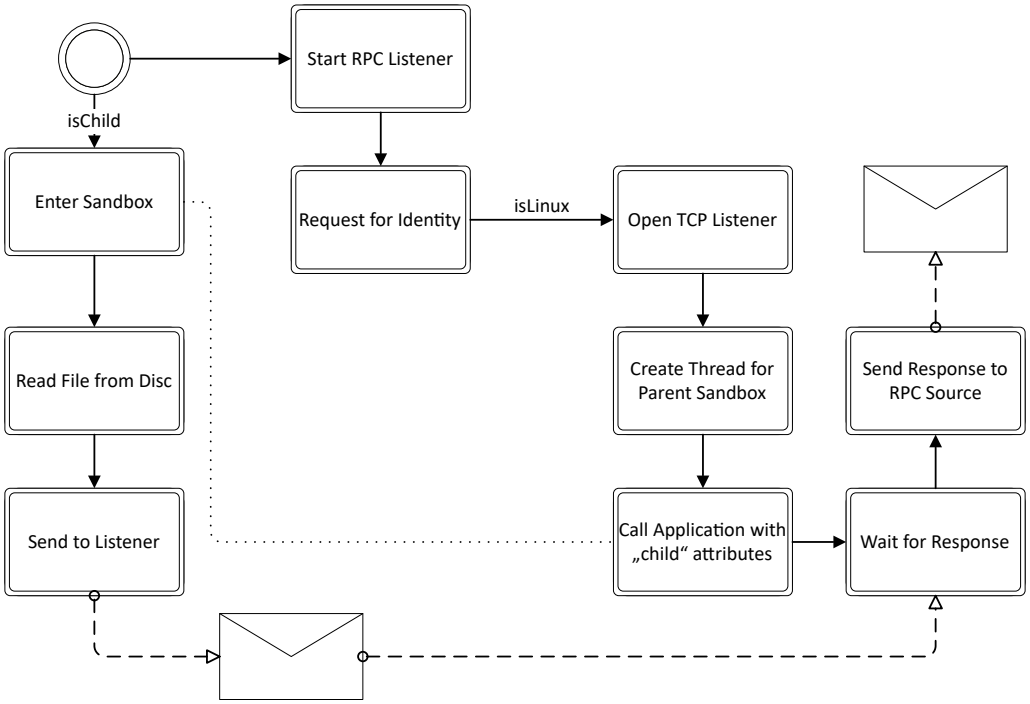


Figure 15: Gaol Sandbox

RPC Listener within the Sandbox

In order to launch the sandbox, we need a parent process that starts a restricted child process. With Gaol, a child process is only allowed to do certain unavoidable operations. Anything the deviates from this standard operation, has to be granted in a whitelist-profile that is defined in the parent process.
Our first approach was to create a sandbox that contains all the identity storage code, besides the few operations that where needed to create the child. Immediately after starting the storage, the child should be spawned. This solution would sandbox everything including the RPC listener and all file access operations.
We found that the TarpC listener could not be started within the sandbox. By inspecting the system calls with `strace` we tried to find the missing permissions to start the listener. We found no missing permissions that would block a socket creation.
In further tests where we extracted the problematic functionality into a separated project, we could not find a way in which a Gaol sandbox allows an incoming connection. Deeper inspections had shown that, even if the socket is created outside of the sandbox, the program will freeze as soon as an the connection is accepted.
We could not solve this problem, and therefore we had to implement a solution where the RPC handling happens outside of the sandbox.

Killing the Child Thread

Furthermore, we came across another problem that occurs when the child thread did block his own thread. This can happen if the thread is waiting for an incoming TCP connection. By killing

the parent thread we expected the child thread to die, independently of its state.
The child did not die with its parent, because it is an independent execution of the application with customized starting parameters, and not a child thread.
The apparent drawback was that the port stayed open and the sandboxed application stayed running.
We tried different sandbox configurations, including different call orders, where we to create the sandbox, but we where not able to solve these issues, either.

About our Result

The in Section 7.4 introduced solution is designed to demonstrate an example, on how to solve the above issues. Nevertheless, this should be seen as prove of concept, not as perfect 'how to' example.
With the presented solution we solved the two main issues: The child will not block any more and will always terminate. Furthermore, we are able to use TCP based RPC and with that combine TarpC and Goal. Most importantly, we show that it is possible to tailor the permissions for a module.
While debugging and testing our solution we also found an issue that we did not create ourself. It is possible to perform any action in non granted directories by accessing it via an relative path. We immediately created an issue report (github:servo:goal:issues:46) on the Github project, but did not receive a response yet.

8 Authentication Sequence

The agent follows a strict sequence of actions and steps that lead to a positive or a negative verification message for the verifier. To provide a rough overview, Figure 16 illustrates the communication events between the sender and verifier, after the digital service discovery.

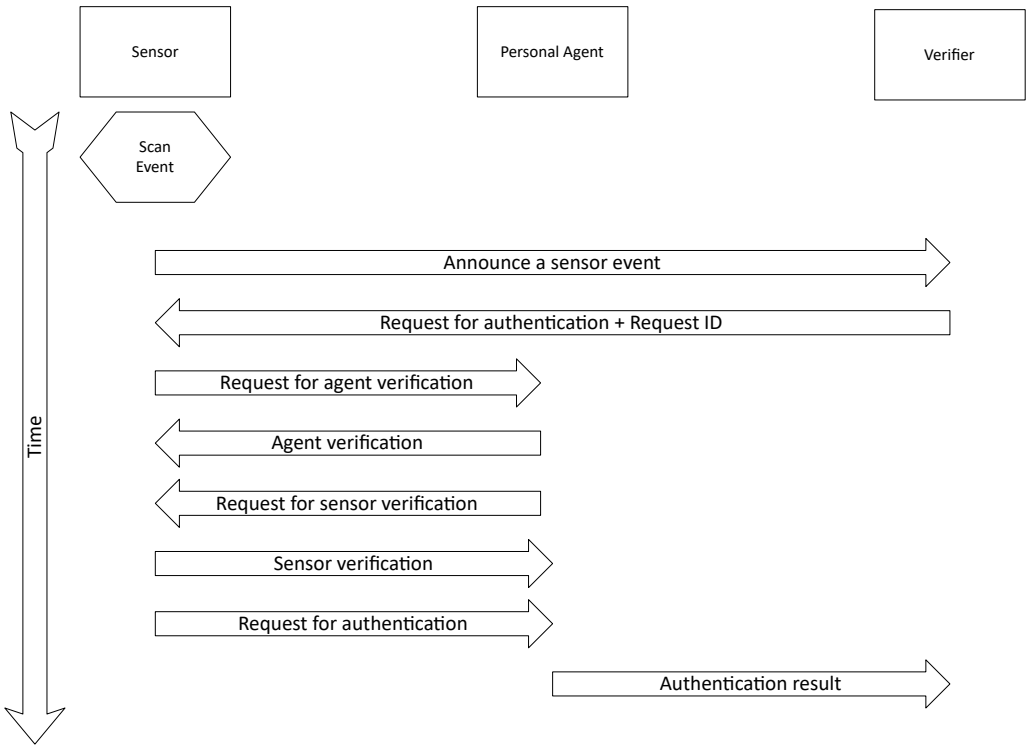


Figure 16: Sender, Agent, Verifier communication overview

The following describes each step of the process, as shown in the figure. Furthermore, the internal data flow for the prototype is described.

Digital Service Discovery

An actual real world process for service discovery is not defined and out of scope for this prototype. The function of a service discovery would cover the connection establishing between the sensor and the corresponding personal agent. We assume that the personal agent is already known in the network and the agent knows all verifiers and sensors. Therefore, in our prototype scenario we assume that scanner, agent and verifier already know each other. Because we skip over that pre-verification step, we act like the agent already knows the ip address of the scanner and verifier.

Request for Agent Verification

The sensor has been triggered with an scan event. He sent an unique id that identifies this event to the verifier. The verifier in turn sends an request for authentication back to the sensor. This request contains an ID that identifies this request.
With the now known ID for this authentication operation, the sensor asks the already known Agent to verify itself.

```
{
"action":"verifyAgent",
"sensor_id":"3B13037C8E7C475D9E14C6F1132A2733",
"sensor_signature": "ssh-rsa 2048 c2:5a:e8:2b:be:c1:5f:ef:94...",
"authentication_id":"DAD95C02443841D09CF0F58FFE6AF0FE",
"timestamp":1518957791667
}
```

- action: current request type definition (verifyAgent).
- sensor_id: static UUID that identifies the sensor.
- sensor_signature: the public key finger print of the sensors public key.
- authentication_id: random UUID to link all messages from one authentication process together.
- timestamp: current sensor time in milliseconds.

Response on Agent Verification

The agent is sending a response to the sensors request.

```
{
"action":"verifiedAgent",
"agent_id":"E64CB51C629C48D8AC331AA714E8AF64",
"authentication_id":"DAD95C02443841D09CF0F58FFE6AF0FE",
"timestamp":1518959089336
}
```

- action: current request type definition (verifiedAgent)
- sensor_id: a static UUID that identifies the agent.
- authentication_id: random UUID to link all messages from one authentication process together.
- timestamp: current agent time in milliseconds.

Sensor verification

Before the agent begins with the authentication process itself, he will verify the validity of the sensor. This step is skipped in the prototype implementation.

Request for Authentication

After a successful agent check and sensor verification, the sensor sends the actual data needed for the authentication process.

```
{
"action":"authenticate",
"data":[120, 76, 44, 68, 201, 115, 235, 117, 64, 141, 42, 131, 155,
↪  140, 46, 200, 13],
```

```
"data_id":"AFA6966DD859482A819EF683F22D6282",
"authentication_id": "DAD95C02443841D09CF0F58FFE6AF0FE",
"sensor_location":"47.871073,13.546948",
"sensor_type":"finger",
"sensor_id":"3B13037C8E7C475D9E14C6F1132A2733",
"sensor_signature": "ssh-rsa 2048
↪  c2:5a:e8:2b:be:c1:5f:ef:94:74:7f:62:ed:86:5a:45",
"service_id" : "passport",
"verifier_ip":"192.168.0.187",
"verifier_port":6789,
"timestamp":1518962878078
}
```

- action: current request type definition (authenticate)
- data: scanned data. Is used as integer array and maps each value to a byte.
- data_id: a unique ID that identifies the current data set.
- authentication_id: random UUID to link all messages from one authentication process together. This id is later used by the verifier to link the result to the request.
- sensor_location: current location of the scanner in longitude:latitude format. This can be used to correlate the location of the request with the persons current location.
- sensor_type: tells the agent, what kind of biometric feature the current data represents.
- sensor_id: again the unique sensor id.
- sensor_signature: again the sensor signature.
- service_id: name or ID for that identifies the requested document.
- verifier_ip: the ip address of the verifier. This is the target to which the agent will send the authentication result.
- verifier_port: open port on the verifier, where the agent can connect to and transfer the result of the authentication process.
- timestamp: current sensor timestamp in milliseconds.

Check - Biometric Futures

Based on the type of biometric features the agent received it will now be compared with a matching template. The biometric module will return the hemming distance between the given template and the given data 4.4.2.

Create - Verification Data

After identification, a response will be created. Independent if the identification was successful or not, a response that will be sent to the verifier is generated. We reduced the response to a boolean information, in order to protect the users identity from the verifier.

```
{
"action":"result",
"authentication_id": "DAD95C02443841D09CF0F58FFE6AF0FE",
"agent_result": true,
"data":  {"content"="value"},
"certificate": "ssh-rsa AA...BQ==",
"signature": "AKSJ3lLNi3nLnE***",
"timestamp":1518963981008
}
```

- action: current request type definition (result)
- authentication_id: the ID which the verifier provided for the sensor, for identification of the request.
- agent_result: a boolean that contains the result of the verification.
- data: a json that contains requested data like name, age ...
- certificate: the certificate from the key store that proves that the person is allowed to do the desired activity.
- signature: the signature / public key of the authenticated person (can be null if not required, or negative authentication result)
- timestamp: current agent timestamp in milliseconds.

Final Step

After the successful transfer of data to the verifier, all connections are closed. The agent returns to the initial state.

9 Implementation Details

In this section we will take a dive into the details of this prototype. The first part is about introducing each module and discussing its final purpose. The second part will provide guideline, how a future developer can start the agent and test its functionalities. The goal is to allow the reader to gain a deeper understanding what we want to show and why we think it should be part of the solution. Furthermore, this could be a reference book in case parts of the code are unclear or not self speaking.

9.1 Overall

The agent is split into multiple parts each of them has one specific task. Furthermore, we use this modules to demonstrate various functionalities.

- Personal Agent Core: Manages the authentication by handling data to and pulling data from modules.
  We want to demonstrate how Rust handles Multitasking.
- Connection: Will listen for incoming authentication requests and can send messages to given targets.
  Basic TCP-In/Out operation will be shown here.
- Identity Storage: Provides the digital identities for the user.
  Parts of the storage will run in a Sandbox.
- Authentication: Is able to sign a given massage on behalf of the user.
  We will use Rust-Crypto to generate signatures. Furthermore, we want to show how to secure RPC with TLS.
- Biometric: Can compare a given data set with saved biometric templates.
  The basic demonstration of RPC.

Furthermore, a Json library, built on top of serde_json, was developed to provide static agent specific templates.

9.2 Personal Agent Core

The core is the centre where all request are processed. We use the core to demonstrate how straight forward threading is in Rust. Therefore, we use multiple threads and implement some common patterns. Also, we show how TarpC requests are sent to RPC providers, implemented in different modules. Figure 17 shows the main thread.

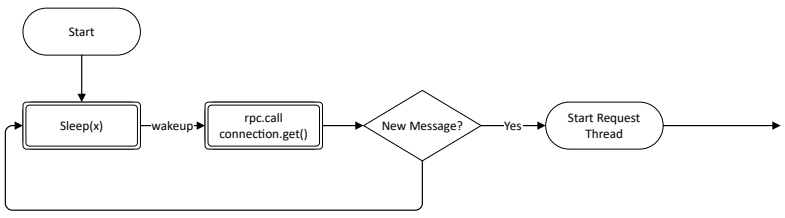


Figure 17: Main Thread

This thread periodically pulls the connection module for new messages. How often this is done is defined in a member variable and sets the minimal time that is needed from start to end of the authentication process. For example if 3 messages have to arrive in order to complete a full request and the pulling time is 10ms, we will need at least 30ms to complete the process. If a new message arrives, the main thread hands this message over to the Request Thread. We discussed in Section 5.1.1, that we choose to use a pull approach over a solution where the connection module pushes the messages towards the core. If the core pulls the messages, the core can control the speed and frequency of authentication requests and therefore the overall system workload at any given time.

The request thread, shown in Figure 18, 19 and 20 handles a message. We are able to handle multiple authentication requests simultaneously, but in the current implementation all iterations are completed faster then the configured pulling time.
In the first step, it will be read if the incoming JSON contains a request for 'agent verification' or 'authentication', the two supported operations.

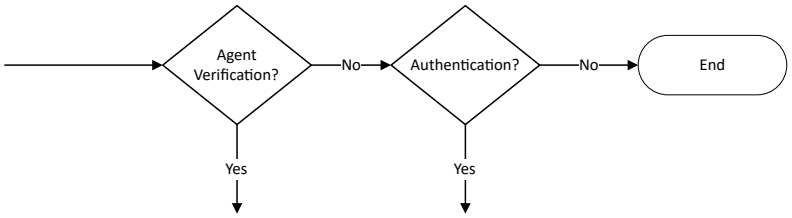


Figure 18: Request Thread

If the 'agent verification' is requested a simple response is generated and the connection module is used to sent it to the scanner. The scanner is defined in the request. Figure 19 illustrates this task.

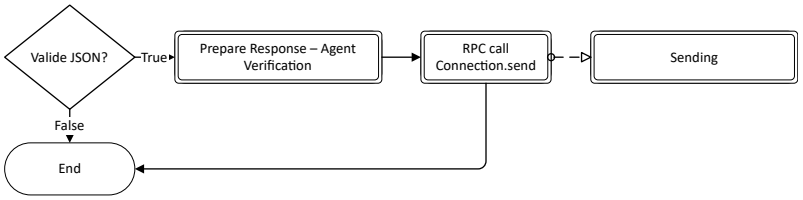


Figure 19: Agent Verification

If an authentication is requested the process illustrated in Figure 20 will be executed.

1. The received JSON is checked if it is valid, in the first step. For that a function, specially designed to check that kind of JSON, is used. The function checks all properties based on hard-coded rules.
2. If the check was successful, the biometric information is used to verify that the received biometric data matches to one of the stored templates. If it does not, a negative result is sent to the verifier and the process ends. Otherwise, the process will be continued.
3. Now the agent tries to get the selected 'service' that, in this implementation, corresponds to an stored identity. If we found an identity that matches our service, we continue. Otherwise, we send a negative result to the verifier.

4. The last but one step is to generate a signature based on data from the currently generated response message.
5. All needed data is collected and the core can create the response message JSON and send it to the verifier.

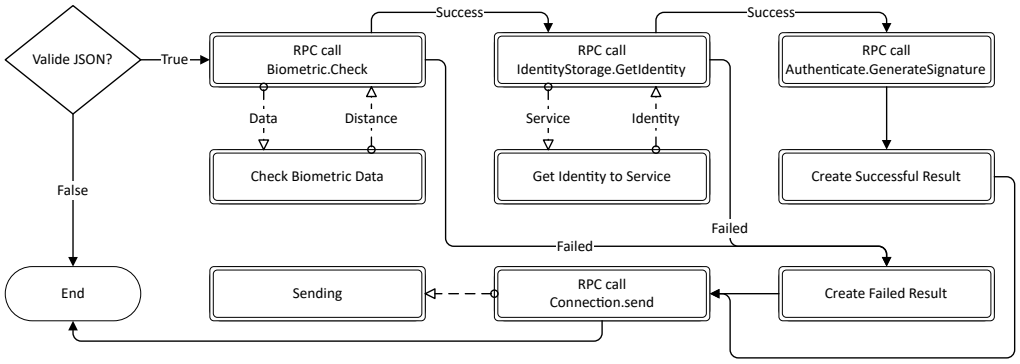


Figure 20: Authentication

Currently the overhead for the multitasking in the core is bigger than the benefit it generates. Besides the purpose of a multi threading demonstration, the idea behind the thread-intensive design is to be fit more load intensive tasks in the modules. In a future implementation, single steps may take a lot of time. This design should make it possible that it is possible to pass threw faster requests.
The core expects the assisting modules to run, if the do not run, it will try to execute a RPC call and, after some time, terminate the request.

Static Variables

The communication ports, where the core will expect the modules, are static. Also, a bit difference - threshold for biometric data as the amount of bits that are allowed to differ between the template data and the scanned biometric data. Furthermore, the query distance is the rate in which new messages will be pulled from the connection module.

```
1 WEB_RPC_PORT : u16 = 5502;
2 BIO_PORT : u16 = 5503;
3 IDENTITY_PORT : u16 = 5504;
4 AUTHENTICATE_PORT : u16 = 5505;
5 SENSOR_PORT : u16 = 5510;
6 BIOMETRIC_THRESHOLD : u64 = 2;
7 QUERY_DISTANCE : u64 = 500;
```

Functions

The core contains a set of functions that are for thread control, RPC calls and sequence control. The set of system operations the core is allowed to execute may be limited to network privileges. The core source consist of the following functions:

- `fn main()` : Starting the main loop that calls `fn check_for_new_messages` after the milliseconds assigned to `QUERY_DISTANCE`.
- `fn check_for_new_messages()` : Opens a connection to the connection-module. If a new message arrived, it will start `fn run_process` by handing over the received message.
- `fn run_process(message:&str)` : If the action is set, it will call the `fn call_corresponding_function` function and hand over the message and action. The action is extracted with `fn get_action`.
- `fn get_action(message:&str) -> Option<String>` : Tries to extract the action. In order to do so, it calls `pa_json::to_untyped_json` to deserialize the message into a JSON. If the action attribute is set, it will be returned.
- `fn call_corresponding_function(message:&str, action:&str)` : If the extracted action is valid, meaning one out of 'Agent_Verification' or 'Authenticate', defined in the `pa_json` module, the corresponding function to process the request is called.
- `fn agent_verification(message:&str)` : The agent has to transmit his ID to the verifier. First, the incoming message is parsed into a strong typed `RequestAgentVerification` JSON. Afterwards the values are checked for limitation with are not covered by the type. If the request is valid, a response will be sent by calling `respond_to_agent_verification`.
- `fn authenticate(message : &str)` : Similar to the agent verification, the incoming message will be parsed into a strong typed `RequestAuthentication` JSON and checked. If the JSON is a valid request, the attached biometric data is compared with `fn compare_biometric_data` Only if the distance between the attached data and a given template is smaller than `BIOMETRIC_THRESHOLD` the request can be valid. If the biometric is ok, the digital identity will be queried with `fn get_ident_for_service`. If a identity could be found a positiv result will be generated with `fn create_successful_authentication_result`. Otherwise, the result will be created by `fn create_failed_authentication_result`. In any case, this result will be handed over to `fn send_to` with the verifier port and ip defined in the request JSON.
- `fn compare_biometric_data(data : Vec<u8>) -> bool` : This will send the transmitted data to the biometric module. Only if the resulting distance is smaller than the `BIOMETRIC_THRESHOLD` a positive response will be returned.
- `fn get_ident_for_service(service_id : String) -> Option<pa_json::Ident>` : A request containing the `service_id` is sent to the the identity storage. If an identity with that ID is stored, it will be returned.
- `fn respond_to_agent_verification(json : pa_json::RequestAgentVerification)` : Thus function will create a `ResponseAgentVerification` JSON and will hand it over to `fn send_to` with the IP and Port defined in `RequestAgentVerification`.
- `fn create_successful_authentication_result( json:&pa_json::RequestAuthentication, certificate:String, ident_content:serde_json::Value)`

`-> Option<String>`: If `fn compare_biometric_data` returns true, a positive answer will be created and serialized, containing the identity that was requested. In case the creation or serialization went wrong, `Option.None` will be returned.

- `fn create_failed_authentication_result(json : &pa_json::RequestAuthentication) -> Option<String>`:
  If `fn compare_biometric_data` returns false, a negative authentication result will be sent to the verifier. The function prepares this response JSON, serializes it and returns it. In case the creation or serialization went wrong, `Option.None` will be returned.
- `fn send_to(to : SocketAddr, message : String)`: If the core wants to send something to a given address, he has to use the Connection Module. This function handles the call.
- `fn get_acceptor() -> TlsAcceptor`: Returns the TLS Acceptor, this is needed to add the TLS layer to the default TCP connection.

Dependencies

The core needs the following external crates to build: tarpc, tarpc-plugins, log, futures, serde, serde_json, serde_derive and rust-crypto. Furthermore, it requires all the RPC definitions for the modules: authenticate_rpc_lib, connection_rpc_lib, biometric_rpc_lib and identity_storage_rpc_li. And finally, the library with the JSON definitions, pa_json.

9.3 Authenticate

The purpose for the authentication module is to demonstrate key and signature generation functionalities. Parts, for example the key storage, of the authentication module can potentially be replaced with secure hardware.
The prototype service listens on Port 5505 and provides a function that allows the core to create a signature for a given byte vector (message). The return value currently is a 64 byte ed25519 signature.
Furthermore, this module is used to demonstrate the TarpC TLS support. The communication between core and module is encrypted.

RPC - Interface

The interface is takes a byte vector and returns a byte vector. Bytes are represented by an unsigned 8 bit integer.

```
1 service! {
2  rpc createSignature(message :  Vec<u8>) -> Vec<u8>;
3 }
```

Static Variables

`PORT : u16 = 5505;`: Port on which the service listens.

Functions

- `fn main()` : Defines the socket address on which the RPC server should listen. It will than call `spawn_server` and take the returned thread handler to attach itself until the application gets forced to stop.
- `fn createSignature(&self, message : Vec<u8>) -> Self::CreateSignatureFut` : Will be called via TLS secured remote procedure call. Is redirecting this call to `fn gen_signature` and returns the result of this call as Rust Result object.
- `fn gen_signature(message : Vec<u8>) -> [u8; 64]` The function implementation is very basic and for demonstration purpose only. It generates a public-private key pair based on the fixed seed value 01255147 (the value is chosen by the author). The private key is used to generate a signature for the given byte / u8 vector.
- `fn spawn_server(address : SocketAddr) -> JoinHandle<()>`: Spawns the RPC Server for the given socket address. In order to spawn the authentication server, the `authenticate_rpc_lib` has to be imported. With this import, the Tarpc-Plugin creates all needed services and server objects. Furthermore, it generates a TlsAcceptor and binds it to its listener. With that acceptor attached, each incoming connection will be encrypted using TLS.
- `fn get_acceptor() -> TlsAcceptor` Reads the X.509 certificate (.p12) and uses a password to extract the Pkcs12 key from it. Afterwards it returns a TLS acceptor which is used to create a TLS secured session for incoming RPC requests.

In sum, the authentication module demonstrates the usage of the rust crypto library and the TarpC TLS. Both technologies might find its place in future development steps.

Dependencies

In order to run the authentication module, the following is required: futures, tarpc, serde, rust-crypto, and the authenticate_rpc_lib where the rpc_lib is the library in which contains the definition for the data exchange interface.

9.4 Biometric

The underlining idea behind the biometric module is that it takes some scanned data from a authentication request and compares that to a saved template. Currently, the implementation is computing the hamming distance between two byte arrays. Afterwards, the distance between the given data and the template is returned.

RPC - Interface

The interface is defined in `biometric_rpc_lib` and provides a check method that returns an unsigned 64 bit value. The function takes a byte vector, that should contain the scanned biometric data and a second parameter probeType that allows to specify, what kind of biometric feature the data represents.

```
service! {
 rpc check(probe: Vec<u8>, probe_type : u8) -> u64;
}
```

Functions

- `fn main()` : Opens the socket for the RPC server. It provides the `fn check()` function. Afterwards it spawns the service and attach itself to it in order to stay alive.
- `fn spawn_server(address : SocketAddr) -> JoinHandle<()>` : Spawns the RPC Server for the given socket address. In order to spawn the biometric server, the `biometric_rpc_lib` has to be imported. With this import, the Tarpc-Plugin creates all needed services and server objects.
- `fn check(&self, probe: Vec<u8>, probe_type: u8) -> Self::CheckFut` : The function takes a vector of 8 bit integers that represents a byte vector. This vector contains the values that where scanned by the scanner and sent to the personal agent to identify the user. On request, the check method will call `fn get_fingerprint_template()` in order to get the stored biometric template from the user. If both vector have the same size, the hemming distance will be computed. In case the size does not match, a maximum distance will be returned. This maximum distance is computed by multiplying the vectors length with 8.
  The parameter probe_type currently is not used. It can be used in future implementations to distinguish between different type of biometric features.
- `fn get_fingerprint_template() -> Vec<u8>` : Fist step in this function is to load the stored template from the hard drive by calling `fn read_fingerprint_setting()` . After a successful load, the content of this file is pushed into a u8 vector which will be returned.
- `fn read_fingerprint_setting() -> Option<String>` : This function opens a hard coded fingerprint file `let file = "../../data/fingerprint_template"` and returns the content as string.

Static Variables

`PORT : u16 = 5503;` : Port on which the Service listens.

Dependencies

Additionally to the standard module dependencies: futures, tarpc, serde, rust-crypto, and the biometric_rpc_lib also the 'hamming' package is used to compute the distance between two vectors of the same length.

9.5 Identity Storage

All digital identities are 'managed' by the identity storage. We selected this module to experiment with Gaol and demonstrate some of its features. The smain requirement for the identity storage module was to limit the folder from which this module can read, when reading identities.

The program takes one parameter, namly the path to the folder where to digital identities are stored. Such a call could be: `identity_storage.exe c:\\digitalidentities`. When the service is called, it will try to find a file that match the given service name plus fixed extension '.ident'. The diagram in Figure 15 shows the standard functionality. The sandbox is only available when Linux is used. We discussed details about that in Section 7.4.
Because there a futures that are only available for Linux, many functions are marked with an attributed that distinguishes the target OS.
If `#[cfg(target_os = "linux")]` or `#[cfg(target_os = "windows")]` is given before the `fn` keyword, the following function will only be build if the operating system is the defined one. The same differentiation is possible for other operating systems like MacOS and Andorid, for example. With that option, we can write multiple function with the same function header for different operating systems.

RPC Interface

The interface for this service is limited to one call where a service name is handed to the module and a string or nothing is returned. As mentioned already in the description of this module, the service_id will be used as the filename.

- `get(service_id : String) -> Option<String>`

Structs

- IdentityServer: The struct that implements `IdentityService` and works as RPC server object.

Static Variables

- `PORT: u16 = 5504` : The port on which the RPC Service will listen
- `INNER_TCP : &'static str = "127.0.0.1:57612"` : The Address for the Parent - Sandbox Child communication

Functions

- `fn main()` : Checks if there is more than one parameter given. If there is a second argument, after the program name, it is the path for the folder that contains the digital identities. After this check, the `fn target_main` will be called.
- `fn target_main()` [Windows]: Creates and starts a event loop and binds the RPC Listener - `IdentityServer` into it.
- `fn target_main()` [Linux]: When called without the child parameter set, it also creates and starts a event loop and binds the RPC Listener - `IdentityServer` into it. When called with the child parameter, it will create a sandbox, read the identities and send the read information to an given TCP Port.
- `fn get(&self, service_id: String) -> Self::GetFut` : This method is part of the `IdentityServer` and is the implemented RPC interface. As soon as a call arrives, it invokes `fn handle_get` .

- `fn handle_get(service_id: String) -> Option<String>` [Windows]: Reads the path where the identities are stored and invokes `fn get_ident` with the service name and the path already concatenated.
- `fn handle_get(service_id: String) -> Option<String>` [Linux]: Opens a TCP listener to which the child thread will report. After that, an thread is created that spawns the parent sandbox and then invokes the identity storage with the child parameter set. Finally, the TCP listener will wait for a response, read it and than return the result to its callee.
- `fn get_ident(path: String) -> std::io::Result<pa_json::Ident>`: Tries to read the identity file from the defined directory, and returns it if its possible to parse it as JSON.
- `fn receive(mut stream : &TcpStream) -> Result<String,&'static str>` [Linux]: Reads an TcpStream till the end and returns the content.
- `fn profile() -> Profile` [Linux]: Creates the sandbox profile. The current configuration allows to read from the path given as first argument and allows all outgoing network connections.
- `fn get_local_server_address() -> SocketAddr`: Shortcut to the local host address as SocketAddr object.

9.6 Connection

This module is by design the only connection with the outside world. Its core function is to accept incoming connections that will send parts of an authentication request. The incoming messages are stored in an message buffer. Via RPC, the core module can pull this messages and handle them accordingly. Furthermore, the connection module provides a RPC function to send a message to an defined receiver.

RPC Interface

The interface defines two functions:

- `rpc send(socket_addr : SocketAddr, message : String) -> bool;`: Can be used to send a message to a given receiver.
- `rpc get() -> String | Message`: Gets, if possible, the received message. For the purposes of demonstration, a custom error type `| Message` is used here. With that it is possible to use self defined error messages. Therefore, we can return not only `Ok`, but also `Err("message")`.

Structs

- `struct IncomingMessageBuffer`: Holds an vector based queue of `String` values. The queue is handled with two implemented functions `fn add(&mut self, s:String)` and `fn get(&mut self) -> Option<String>`.
  It is part of the `static MESSAGE_BUFFER` that allows to add and pull received messages from multiple threads.

- `struct TCPModule` : Implements the ConnectionService and provides the functions defined in the RPC definition. It implements the `fn send` and the `fn get` functions. We will read more about them in the section about functions.

Static Variables

```
lazy_static! {
        static ref MESSAGE_BUFFER: Mutex<IncomingMessageBuffer> =
 ↪  Mutex::new(IncomingMessageBuffer::new());
}
```

We use a `lazy_static` to be create a dynamic data structure, that is wrapped in the `IncomingMessageBuffer` structure. The structure again is wrapped with an mutex, that enables us to access this object without an `unsafe` block from multiple threads.
Furthermore, there are the two ports where the connection module listens `static TCP_PORT = 5501` for incoming connections from the internet and the port for incoming RPC requests `static RPC_PORT = 5502`.

Functions

- `fn main()` : Calls a `fn start()` , a function that will spawn both, the TCP listener and the RPC server. Also a synchronous channel is created that saves received messages into the `MESSAGE_BUFFER`. The messages will be generated in the `fn handle_client()` method, that is called for each incoming connection.
- `fn start(tcp_server_addr : SocketAddr, rpc_server_addr : SocketAddr, send : Sender<String>)` : The function spawns two listener, one for TCP and the other for the RPC requests.
- `fn handle_client(stream : TcpStream, send : Sender<String>)` : All successful received messages that are passed back by the `fn receive()` function are sent to the main thread via the 'send' object.
- `fn receive(mut stream : &TcpStream) -> Result<String,&'static str>` : Working with a 256bit buffer this function will listen for new characters in the incoming stream. As soon as the message is complete, it will create a `String` message out of it and will return the successful Result to the `fn handle_client` function.
- `fn send(&self, socket_addr : SocketAddr, message : String) -> Self::SendFut` : This function will open a new outgoing connection to the provided socked address. It will than serialize the given message and send it to the receiver. The returned value contains true or false depending on the success of the connect and send operation.
- `fn get(&self) -> Self::GetFut` : This function locks the `MESSAGE_BUFFER` and than returns the first element from the queue.

9.7 JSON Handling

JSON is the main data format that is used for the communication between scanner, verifier and agent. Therefore, a solid library that handles these JSONs is obviously very important.

We developed a tailored wrapper for SerdeJSON that is located in `\libs\pa_json`.

Static Variables

The customized JSON handler provides also a single point for the possible actions. Each of these actions corresponds to an predefined JSON structure.

```
pub static ACTION_NAME_AGENT_VERIFICATION : &'static str =
 ↪  "verifyAgent";
pub static ACTION_NAME_AGENT_VERIFIED : &'static str =
 ↪  "verifiedAgent";
pub static ACTION_NAME_AUTHENTICATE : &'static str = "authenticate";w
pub static ACTION_NAME_RESULT : &'static str = "result";
```

Structs

All structs in this library represent a defined JSON that is either received or sent. The content of each JSON strictly follows the definition from Section 4.

- `pub struct RequestAgentVerification`: Implements `UntrustedJson` and therefore the `check()` method is available for an object of this type. `Fn check()` for that type validates that the given `sensor_id` is a valid UUID.
- `pub struct ResponseAgentVerification`: Implements the `SendableJson` trait. We explained already, that this trait provides us with an special `to_string()` method that serializes the given JSON and provides us with a String value, that can be sent to the receiver.
- `pub struct RequestAuthentication`: Implements `UntrustedJson` and therefore the `check()` method is available for an object of this type.
- `pub struct ResponseAuthentication`: Also, implements the `SendableJson` trait. Calling `fn check()` for that type, is validating that sensor_id and authentication_id are valid UUID values.
- `pub struct Ident`: Also, implements the `SendableJson` trait. The 'Ident' JSON differers from the others with the dynamic field `content`. This field has the type, defined by serde_json, `value`. The value type can contain any valid JSON structure. Being unable to define how a future digital identity will look like, we use the value type to include any given identity into the JSON.

```
pub struct Ident{
 pub content : Value,
 pub sha512 : String,
 pub rsa_public : String
}
```

Traits

Following public traits are provided:

- `pub trait UntrustedJson`: All incoming JSON have to implement this trait. With it, a function `fn check` has to be implemented for that type. The implementation verifies the data beyond usual type bounds. Its implemented for RequestAgentVerification and RequestAuthentication.
- `pub trait SendableJson`: Is a wrapper that provides the `fn to_string` method for JSON objects. Its implemented for ResponseAgentVerification, ResponseAuthentication and Ident

Functions

- `pub fn to_json<T:DeserializeOwned>(input : &str) -> Result<T, Error>`: Uses `serde_json` to parse a string into one specific JSON. If they do not match an error will returned.
- `pub fn to_untyped_json(input : &str) -> Result<Value, Error>`: Uses `serde_json` to parse the given string to an JSON without any given template. If the value is a valid JSON it will be returned, otherwise the return value will be the parse error.

9.8 Running the Personal Agent

In this section we want to present a way how the user can run and test the personal agent.

9.8.1 Prerequisites

The tarpc-plugins use `#![feature(plugin_registrar, rustc_private)]`. `Plugin_registrar` allows to interact with rustc directly, for example, to attach a Rust macro. `Rustc_private` is a library feature that allows us to use private compiler features. Most of them are marked as unstable.
As a result, it is currently not possible to build the Agent with the stable toolchain. In order compile the personal agent, we have to change to the current nightly toolchain. The most convenient way is to use rustup to download the current nightly version and set it as the default:

```
1 rustup install nightly
2 rustup default nightly
```

If working with the nightly version is a problem for other projects, there are also options to overwrite the default compiler for single projects. When rustc and the rest is installed, cargo will be available and we can use it instantly to compile Rust programs with `rustc` or `cargo build`. The source code for the personal agent can be downloaded from `git@git.ins.jku.at:proj/digidow/agent-rust.git`.

9.8.2 Compiling

The sequence in which we built the individual parts is not relevant. We have to navigate in each of the following directories and run `cargo build --release` to build a current optimized binary.

- /modules/authenticate

- /modules/biometric
- /modules/connection
- /modules/key_storage
- /core

After successful building all parts, we can navigate into the target directory and run the corresponding part. An convenient alternative is to open multiple terminal windows and run `cargo run --release` instead of `cargo build --release`. This will immediately start the binary after building it.

9.8.3 Problems

We encountered common problems during development and testing:

- If there are problems with openssl under Linux, it might help to (re)install `pkg-config`.
- Tarpc-plugins cant find '.ident'. Use `rustup update` do update your nightly installation.

9.9 Integration Test

There are multiple ways to test a given Rust program. One can simply use the built in test function that come with Cargo. We would provide special annotated modules and functions and test the software with them. The drawback, as we see it, is that these tests are manly designed to test libraries, not executables. Furthermore, use Rust to test Rust, will reuse the same libraries for the program and the test and with that, ignore problems that may occur when we use other TCP sockets from other languages.
Therefore, we decided to use two Java programs to test the default behaviour. For that, the SimpleScanner and a SimpleVerifier are provided. The main purpose of this duo is to test the Personal Agent implementation by sending and receiving one or multiple authentication messages. Figure 21 and 22 illustrate the inner mechanic of this two programs.

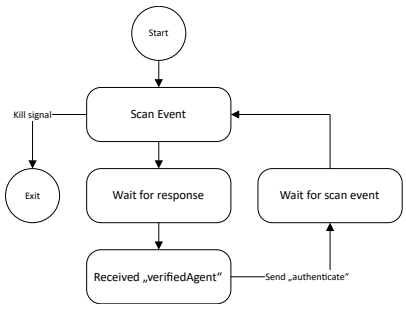


Figure 21: Scanner process

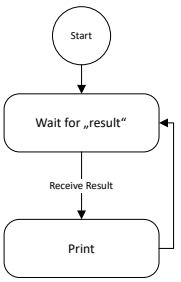


Figure 22: Verifier process

This test demonstrates that the concept and the developed functionalities of the prototype work. To generate the response, the agent has to communicate with each module and has to work through the whole authentication process.

10 Evaluation

The introduction in Section 1.3 defined targets for the introduced prototype. The targets are:

- Implement a personal agent prototype in Rust.
- Show basic functionalities and allow the evaluation Rust as the language for the personal agent.
- Set cornerstones for further development with good an expendable architecture.

The following sections will cover these three bullets and evaluate if they where achieved.

10.1 Implement a Prototype

Our goal was to implement a personal agent prototype in Rust. Furthermore, to obtain an impression, if Rust is a good language choice to implement this project.
Details about the implementation where already discussed in Section 9. In this evaluation we will take a closer look on the agents behaviour.

10.1.1 Agent in Numbers

The following table provides numbers to gain an overview over the projects complexity.

Short	#	Details
Runnable Binaries	5	Core, Authenticate, Biometric, Connection, Identity Storage
Libraries	5	Authenticate RPC, Biometric RPC, Connection RPC, Identity Storage RPC, pa_json
Lines of code	1087	Core (318), Authenticate (77), Biometric (69), Connection (181), Identity Storage (266), Authenticate RPC (12), Biometric RPC (12), Connection RPC (15), Identity Storage RPc (15), pa_json (122)
Used libraries	10	futures, hamming, log, lazy_static, rand, rust-crypto, serde, serde_Json, serde_derive, tarpc tarpc-plugins, UUID

Table 2: Agent in numbers

10.1.2 Startup and Shutdown

The current solution as a whole consist of multiple standalone binaries. The sequence in which the modules are started, and the current execution state of modules, is currently not controlled. Therefore, it is possible that problems occur when a required module is not running at a given time. This would be a major drawback of the proposed architecture for the prototype, and may make the it not practical to use.
The following will show, that the starting sequence is irrelevant. From that we see, that the each module in this architecture is robust enough, to operate on its own. Nevertheless, an authentication is only possible, if all modules are operating normally.
The following modules exist:

1. Core
2. Connection

3. Biometric
4. Identity Storage
5. Authenticate

When we restart the agent we first start all the modules and afterwards start the core. In that order, it is guaranteed that no module is needed before it is available. We will now take a look at different starting sequences and corner cases.

Message arrives before the Core is running: We start all modules, send a message and afterwards start the core. The result is an controlled behaviour, where the system can recover as soon as the core is started again. As long as the connection module is not restarted, the core can pull the request as soon has he is running. If the connection module would be restarted before the core pulls the message, the messages will be lost. The prototype does not persist its message buffer.

Core and Connection module are running: We start core and connection module and send a message. The core will try to query data from non running modules.

```
1 Action: verifyAgent
2 Successful send (Target: 127.0.0.1:5510):{"action":"verifiedAgent",
   ↪  "agent_id":"...", "request_id":123456, "timestamp":1527523462}
3 Action: authenticate
4 No connection could be made because the target machine actively
   ↪  refused it. (os error 10061)
5 Successful send (Target: 127.0.0.1:6789):{"action":"result",
   ↪  "authentication_id":"...", "agent_result":false,"data":{},
   ↪  "certificate":"", "signature":"", "timestamp":1527523463}
```

Line 4 in the above core console output gives us a hint that one module could not be reached. The default behaviour for that is to send a negative authentication response. The connection module is still running and so we are able to send it.

Connection module is not running: If the connection module is not running there is no way for the personal agent to communicate with its surrounding world. Our Java Scanner crashes with an exception because it is not possible to connect to the agent.

Verifier is not reachable: We start all modules and the scanner sends an authentication request. The result should be sent to an verifier that is currently not available.

```
1 Failed to send (Target: 127.0.0.1:6789):{"action":"result"
   ↪  ,"authentication_id":"...", "agent_result":false,
   ↪  "data":{},"certificate":"", "signature":"",
   ↪  "timestamp":1527524266}
```

The above console output tells us that the, in this case negative authentication result, could not be sent. After this failed attempt, the core will recover.

As mentioned at the beginning, we see that the prototype already proves that a robust solution is possible with the current architecture.

10.2 Accomplished with Rust - Evaluate Rust

What we saw until now is that we where able to actually implement a prototype with Rust. Further, we want to reflect an evaluate how Rust and the environment supports us on the following set goals. The goals where:

- Verify, by following to the requirements, that a Personal Agent can be implemented in Rust.
- Show that Rust and its surrounding frameworks provide tools that enables us to include security relevant features.
- Set a good start point for future developments in Rust.

The key points that we implemented in this prototype are:

- An event based Rust application.
- A Scalable thanks to easy multi threading.
- A flexible module design.
- Secure communication between the modules with TLS secured TCP-RPC.
- Using state of the art JSON for data abstraction.
- Secure handling of untrusted data.
- Developed a future-oriented project design.
- Using a Rust sandbox.

10.2.1 Rust Future - Multi Threading

The core application and its communication with the modules represents a good multi threading example. If multiple messages processed at the same time, the core will spawn a thread for each of them, as shown in Figure 17. Depending on the kind of request, each thread will spawn its own event loop to handle the RPC communication.
Rust reliable ensures, that no data race can happen. Moreover, it would have take an deliberate design decision to allow data sharing between these threads. Therefore, by design and additionally controlled by the Rust compiler, it is ensured that each request is handled isolated from the others.
In terms of performance, a pure sequential implementation would reach the same or even a better score. We assuming that currently the overhead that comes when spawning a new thread slows down the application. However, this implementation represents a prototype that should demonstrate the possibilities with Rust and to create a layout for future development. Thinking about future extensions in the external modules, where cryptographic procedures and database request are required, this parallel non blocking implementation will be the key for a static performance guarantee.

10.2.2 A flexible Module Design

With the RPC architecture, that works with separated interface libraries, we provide an fixed and simultaneously variable way of working on separated modules. One could simply take one interface and replace the underlining module. For the core it is transparent what happens in this modules, as long as the interface restrains are keep. The major drawback on the current implementation is, that the core can only connect to one module. Meaning by design it is currently not support to have multiple modules of the same type. The result is that the whole functionality has to be kept in one binary. Separating the same module in multiple binaries could be beneficial when, for example the requirement of a fingerprint and a separated 'voice' biometric module

comes up.
Nevertheless, the design currently provides the benefits that one gains by working with clean interfaces. Following this idea, should make it possible to separate the whole personal agent project into individual independent groups.

10.2.3 Memory Management Pays Off

Creating objects and not care about what happens to them after they are not needed any more, is something that was nearly exclusively reserved to languages with garbage collector. Rust allows us the same careless usage of heap objects. We already discussed how the compiler extends our code and frees the memory, as soon as an variable goes out of scope.
Thanks to this helping mechanism, a very fast and fearless prototyping is possible. Developer of all levels are able to contribute, without having to fear that memory security could be broken, as long as they do not use `unsafe` blocks. This potentially reduces debug and code review time.
Moreover, it allows an overall small memory footprint, without the dangers of memory leaks. In the exact moment where a variable is out of scope, the memory is available without an potential interruption by an memory management system.

10.2.4 Environment is Ready for Production

Intuitively we assume that the environment that comes with an language is an critical factor for its success. Luckily, the Rust community shares this impression and set an early and deep focus on that. Rust comes with many tools that help the developer being productive and not wasting time with updates, build scripts, dependencies or similar important things. Especially system languages tend to provide a certain degree of freedom, that comes with some complexity for first-time user.
Some important tools we used are:

- Rustup: To install and update Rust versions
- Cargo: To create and build a project. It also downloads dependencies and builds them. Cargo allows to run tests, benchmarks and examples.
- Cargo-Edit: Is a extension for Cargo that allows update and add dependencies [46].
- Visual-Studio-Code: Is one of the supported Editors and already offers Rust extensions. Alongside with code formatter and syntax highlighter we are able to see compiler errors directly in the source code.

Overall, in our opinion the Rust environment is ready for projects of any kind and size.

10.2.5 Basic Security RPC + TLS

The available security relevant features that Rust offers, are another important aspect to consider, if Rust should be used for this kind of application. We secured the RPC communication between the modules with TLS, to demonstrate this basic functionality.
As already mentioned, it is possible to use TLS secured TCP channels for RPC communication. We now take a deeper look at this communication and see if TLS is in fact active and which information is still visible.
In order to be able to have readable data sent, we change the data that we use for a signature from

the public key to the actual data that is contained in an stored identity. Figure 23 shows that we are able to catch and read all the content that is sent to our module. Certainly, the same thing is possible with the generated signature that is sent back from the module to the core.

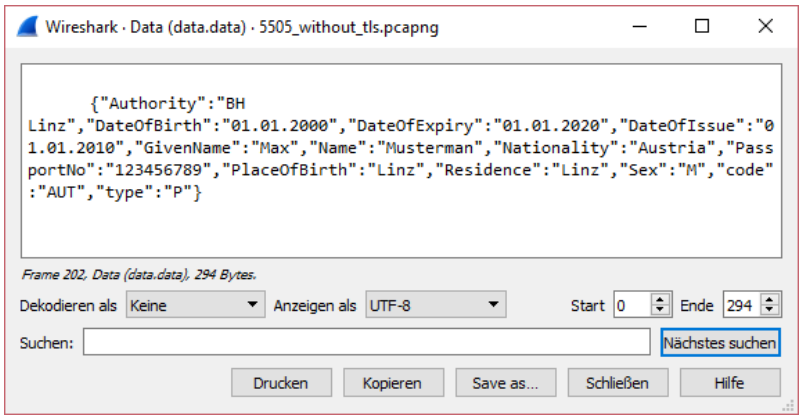


Figure 23: Core sends Passport without TLS

After activating TLS we immediately see that it is actually working. The first thing we can observe is the additional overhead that is added for the SSL handshake. After the successful handshake, we the protocol is switched from TCP to TLSv1.2. Figure 24 shows this communication.

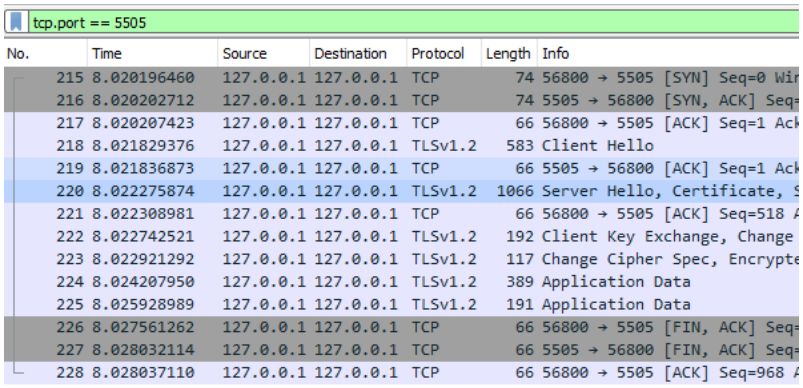


Figure 24: Communication with TLS

Proving that the message is actually not readable, would be hard and out of scope. Nevertheless, we see from the intercepted message that the data, sent from the core our RPC server with active TLS, seems to be encrypted. Figure 25 shows the passport content with active encryption.
We demonstrated that the TLS support of TarpC is in fact working and configured an example for our prototype. Furthermore, it is shown that Rust natively supports basic cryptographic functionalities.

10.3 Set Cornerstones

With the prototype we also set a possible direction for future development. Therefore, the architecture and used technologies where selected carefully. After all, we see the architecture as easy expendable and flexible enough, to be the template for the future.

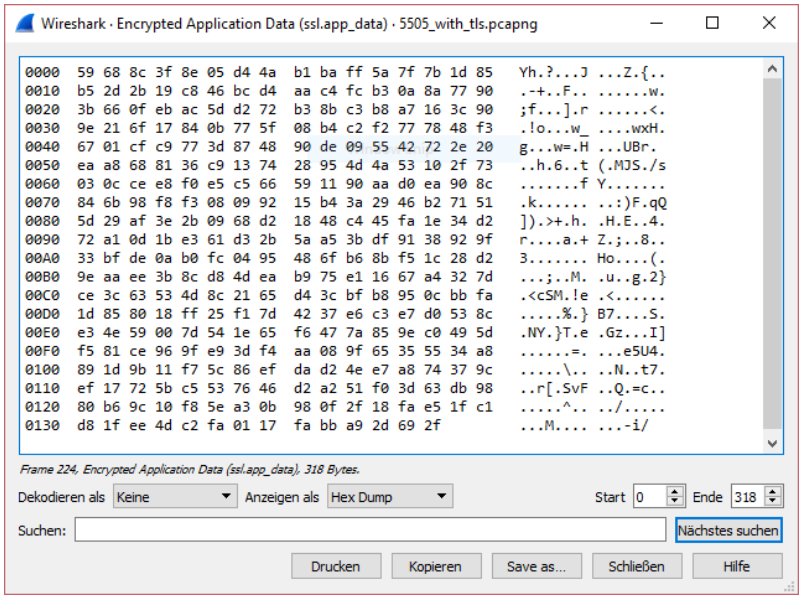


Figure 25: Core sends Passport with TLS

10.3.1 Architecture Benefits

The central benefit is generated from the evolved architecture. We already introduced the final solution for the module system architecture in Section 5.1.1. The implemented solution gains security benefits and advantages for future development.
Each module is its own project with a once defined communication interface. In a future process design of this interface can be refined. The corresponding module can be adapted to this new interface, independently from all other modules. We had the self set rule, that all communication between the modules have to be triggered by the core module. Therefore, there are no dependencies between the individual modules.
With a password protected certificate for TLS-RPC communication, it is possible to ensure that only trusted modules can use the RPC channels.
Moreover, the separate project enables multiple team members to work individual without impact on the work of others. A benefit especially during the first development iterations with frequent changes.
Running each module as independent task also adds another layer of security and flexibility. The operation system ensures that each task has its own execution environment. With that, the likelihood of exploiting the whole system, with an security problem in one module, is reduced. Moreover, we can monitor, debug and analyse single modules without influencing or restrict other parts.
Similar to be previous benefit, it also allows us to update modules without having to restart the whole system.

10.3.2 Basic Threat Handling - Possible Threats

Be aware of as many threats as possible is an important requirement to ensure a certain level of safety. We make use of the CIA principles, as typical information security guideline, for our evaluation. CIA stands for confidentiality, integrity and availability [1]. All three are important

security aspects to be considered for our agent.

Availability: The agent has to be reachable when a request should be handled. An unreachable agent may be interpreted as an forgotten identity card or ticket, which could lead to penalties. Furthermore, if the agent fails to do what he is supposed to, people will lose trust in the system. This makes an attack on the availability an interesting target if someone who wants to sabotage peoples mind by creating negative impressions. Making it impossible to verify an identity can also be used to harm targeted persons, when done in delicate situation, like leaving a country or boarding a plane. Following are some examples of what could course a problem for the agents availability:

- Unreachable while running by overload.
- Unreachable by forced shut-down.

In our following tests we are going to slip into the role of an potential attacker and try to overload the agent or force a shut-down.

Confidentiality: A central target of the decentralised agent solution is to provide each person a trustworthy manager for his identities. If the agent relinquishes any information it is not supposed to, it would be a break in confidentiality. This could mean that he gives away to much information during an authorised and valid communication or to an unauthorised third party. The prototype currently does not implement any counter measurements for that. The focus was on the implementation itself, not the communication with the external world. So, testing the following scenarios would be pointless:

- Reading identities that a service is not supposed to.
- Getting any information as response that is confidential without authorisation.

On the other hand, we implemented some countermeasures to prevent undefined behaviour.

- Reading files that are not supposed to be reed
- Using accepted data to harm the system

With our sandbox, we try to omit any unsupported file reads and with the strict JSON handling, we try to limit the zone where untrusted data is used. We are going to take a look and test if we can verify that this is working as expected.

Integrity: In the context of this personal agent it means an assurance that the communication and stored data can not be modification, without our intention. In cases where something seems odd, because of unexpected or intended mutation of data, the system has to be able to detect this. Meaning, the agent has to protect and detect unauthorized modifications [13].
The prototype does not verify the content of incoming data and therefore, there is no integrity validation at this point. Also, stored files, which are also considerable as untrusted data, are not verified before they are read. But, outgoing messages that contain an identity, also own a signature that should be used to verify the contained information. We are going to check how this could be done as a verifier.

- modification of identity information between agent and verifier.

The following section focuses on the above discussed possible vulnerabilities and demonstrates how they are prevented.

10.3.3 Basic Threat Handling - Tests

We want to inspect scenarios in which we try to make the agent unavailable.

Connections

Test case	Open a maximum number of connection
Target	Make new connections impossible
Description	We will use a JAVA application to open as many connections as possible. To get this number, we will do two runs, where we will find the amount of allowed connections in the first run. The second run will use this amount, open it and wait till the application is shut down. The default JAVA - Scanner will be used to test, if its still possible to open connections from another application.
Experiment	We will use the Java program - Connection.MultipleConnections and the default scanner.
Expected Result	We expected that there is a limit per client application and the server should still be capable to accept new connection from other applications.
Result	We are able to open 129 connections, before a new connection would result in an connection error. When inspecting the status we are able to confirm 129 established connections between our test application and the connection module. If we then try to use our scanner, we see an error that tells us that the connection was refused. Therefore, it is possible to make our agent unavailable by keeping connections open. This circumstance threatens the availability of the agent, and should be considered in during future development.

Table 3: Multiple Connections

Endless Data

Test case	Sending endless data
Target	Filling up the systems memory to the limit be sending an endless stream of data
Description	We want to try how the connection module behave when an never ending stream of data is sent. We will use a JAVA application that generates an endless stream of random characters and send that via an open connection to the module.
Experiment	We use the Java program Connection.BigData and the Connection Module. We will observe the operating systems memory usage.
Expected Result	We expect that there is some sort of built in limit or limited connection time.
Result	In the final version the prototype limits the maximal number of characters that will be accepted. The threshold is set directly in the connection module with the constant value static MAX_RECEIVE_CHARS. Therefore, it is not a problem any more. At first, it was possible to fill up the memory. There where no limits or time restrictions. During the development an unexpected security feature was the duration of the 'is a valid JSON' check. It made it infeasible, because each MB of memory nearly 30 seconds. We fixed that, to increase the performance.

Table 4: Endless Data

10.3.4 Basic Threat Handling - JSON Handling

The data for the communication between agent and the outside world is formatted as JSON . The following shows, that basic JSON malformations are detected and appropriate handled.
Each test case is documented explained.

Test case	The name or short description of the test case. In each scenario we will send JSON data.
Target	The defined target before the test case was implemented. In general we want to fore the agent into unexpected behaviour by sending porously malformed JSON or JSON with unexpected content.
Description	A more detailed description for each test case, defining what we expect to observe. Basically, we want to observe if the agent catches invalid input and how they are handled. In any case we want to prevent any unexpected behaviour that would probably open the door to an exploit.
Experiment	For each test case we describe, what will be sent towards the agent. We will use the JAVA application 'Connection' to send invalid data sets with the goal to test the following cases: 1. Malformed JSON 2. No action 3. Invalid action 4. Valid action, invalid fields 5. Valid action, valid fields, invalid content
Expected Result	We expect the agent to catch all cases and end the authentication either by ignoring the request or sending a negative authentication response.
Results	Are shown below, for each scenario

Table 5: JSON Handling

Test case	Basic JSON
Target	Finding malformed JSONs that are accepted by the agent.
Description	We prepared a set of data against which we test our agent. This set contains one valid and multiple malformed JSONs.
Experiment	The JSONs shown in the listing below will be sent to the agent. This is done with the Java application Connections.InvalidData, where the payload is defined. We will observe the agents output to see, how that input is handled.
Expected Result	We expect the JSON parser to catch all cases and to not except any of tested examples
Results	All malformed examples where detected. As soon as the stream is closed, the connection module assumes that the received data has to be a valid JSON. If it is not, the data is dropped with the error message 'Not a JSON but Stream ended'.

Table 6: Basic JSON - Test Case

The following list of JSON examples, was sent for the basic JSON test case, described in Table 6.

accept `{"action" : "test"}`

deny `{"action":"test"2 }`

deny `{"action""test" }`

deny `{"action": }`

deny `{"action":" }`

Test case	Valid JSONs with an invalid action.
Target	Finding a action the agent accepts JSONs without an applicable JSON template.
Description	We prepared a set of data against which we test our agent. This set contains JSONs without an action tag and with invalid action value. These JSONs are well formed JSONs and they should be accepted by the connection module.
Experiment	The JSONs shown in listing below will be sent to the agent. This is done with the Java application Connections.InvalidData, where the payload is defined. We will observe the agents output to see, how that input is handled.
Expected Result	We expect the core module to catch all cases and to drop the request.
Results	The connection module accepted the JSONs because they are formally valid data. As expected the core module immediately dropped the requests with an error messages 'Unknown action' and 'No action'.

Table 7: Invalid Action

The following list of JSON examples, was sent for the valid JSON invalid action test case, described in Table 7.

deny `{"action":"hack"}`

deny `{"noaction":"authenticate"}`

Test case	Valid JSONs, valid action and invalid field.
Target	Find a way to trick the agent into accepting a invalid data.
Description	We send a JSON that will trigger a certain procedure, when a valid action is called. Therefore, the stored JSON template will be applied and used to parse the data into an object. We will check, if the agent accepts JSONs that do not match with the target structure.
Experiment	The JSONs shown in the listing below will be sent to the agent. This is done with the Java application Connections.InvalidData, where the payload is defined. We will observe the agents output to see, how that input is handled.
Expected Result	We expect the parser to detect all cases and return an error as Result.
Results	The matching action allows us to apply a structure as template for the incoming JSON. As expected, we log error messages like: 'invalid type: integer '42', expected a sequence at line 1 column 38' when we try to parse the incoming stream into the matching structure.

Table 8: Valid Action, Invalid Fields

The following JSON example, was sent for the valid action, invalid field test case, described in Table 8.

deny `{"action":"authenticate", "data":1234 }`

Test case	Valid JSONs, valid action, valid fields and invalid content(additional constrain test)
Target	Find loop holes for data that can not be processed.
Description	After the template from the previous test was successfully applied to the JSON a custom check() method is called. We will try to find ways to insert data into our agent that potentially harm it or at least create some undefined behaviour.
Experiment	The JSONs shown in the listing below will be sent to the agent. This is done with the Java application Connections.InvalidData, where the payload is defined. We will observe the agents output to see, how that input is handled.
Expected Result	We expect that the custom check() method allows the agent to detect all unusual input.
Results	The invalid timestamp was detected by the check method. Further processing was denied. The timestamp is currently the only data that has to pass logical tests while the JSON is processed.

Table 9: Valid Action, Valid Fields, Invalid Conten

The following JSON example, was sent for the invalid data constrain test case, described in Table 9.

```
{
"action":"verifyAgent",
"sensor_id":"16e04827-61f3-488e-afe7-71f062b687a1",
"sensor_signature":"asndfkahkdfnakfnaurojn38u498jofsr",
"request_id":123456,
"timestamp":1234
}
```

With all these basic tests we have shown that the concept behind the JSON based communication is fully functional and able to detect and handle problems with incoming data.

11 Discussion

This section provides details about the following topics:

1. The flow of untrusted data. Our thought when we had to handle JSONs from the outside.
2. Current speed measurements. We examined behaviour details about the message polling, by measuring the total response time.
3. Why Rust and Gaol, instead of Java and Docker.

These topics collected in this separated section are additional insights into the thoughts behind decisions that lead to this prototype. Furthermore, some of the presented results are an additional product and not necessarily related to the previous set targets.

11.1 Use of Untrusted Data

In our opinion, the handling of data that is provided from third parties are a the major risk. We inspected the agent and searched all methods where untrusted data is used, in order to focus on these functions and there behaviour.
We assume that, after the JSON passed all the tests discussed in Section 10.3.4 it is safe to use. Figure 26 illustrates the way, a message is handled before it reaches the point where we consider it as safe.

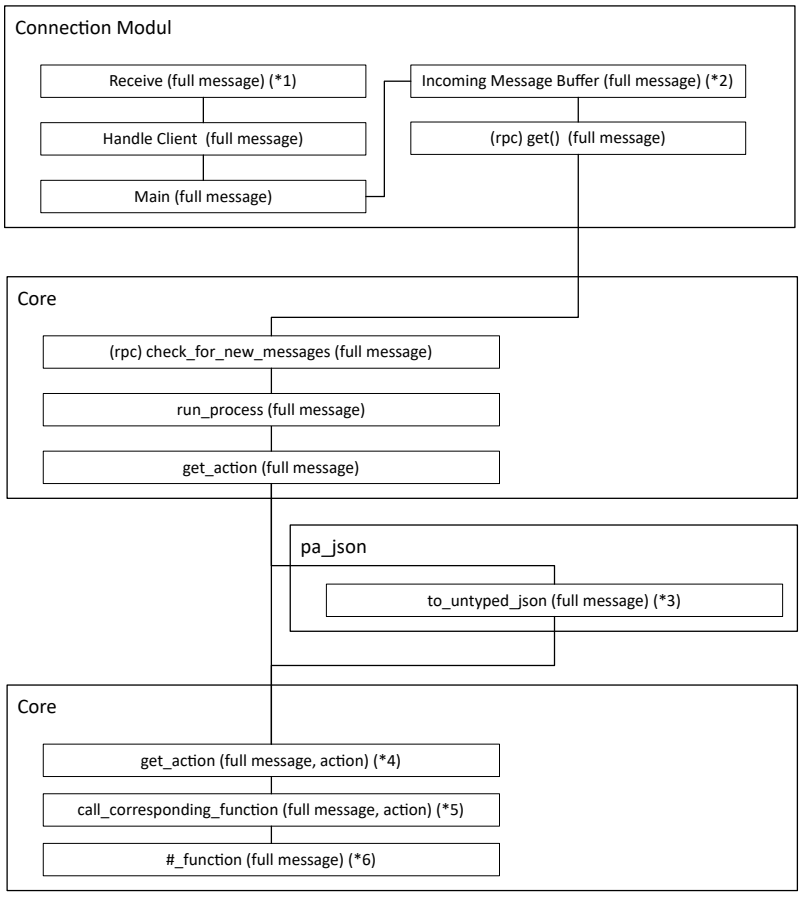


Figure 26: Untrusted Message Flow

The parts that are marked with a * are considered as potential dangerous, because there a read operation is executed. Table 10 provides a short descriptions about where and for what the data is used. Furthermore, Table 11 shows how we secured the part.

***1** Receive Message	Writing a TCP Stream into a string
***2** Incoming Message Buffer	Writing the message into a string buffer
***3** To Untyped JSON	Parsing the message into an JSON to read the action
***4** Get Action	Calling the Untyped JSON and working handling the read action
***5** Call Corresponding Function	Calling a function based on the action
***6** #_Function	Here we parse the message to a typed JSON and check it

Table 10: Functions with Untrusted Data

***1** Receive Message	Limited to an given amount of chars
***2** Incoming Message Buffer	Not protected, strings of limited size are stored
***3** To Untyped JSON	Only valid JSON text-content is accepted
***4** Get Action	No further protection needed.
***5** Call Corresponding Function	We do not print, nor save the action. We just call it in an match statement and afterwords drop it.
***6** #_Function	We do not use the action information directly. Rather we parse the JSON based on the called function and its associated expected object. Moreover, if we can not link the action to a predefined function, the check will fail.

Table 11: Handling of Untrusted Data

What we show with that information is how we use untrusted data, but thought carefully about how to handle it. By limiting the size and using the input only for decisions, we tried to limit the impact such unchecked data may have. However, we use the content of the JSON, after its check, as if it is safe data. Without a doubt it is necessary to increase the tightness of the functions that verifies the content. Moreover, to introduce many plausibility checks to verify if, for example, a given IP address is the real verifier that corresponds to a given scanner.

11.2 Speed

We want to measure how fast our current prototype can handle a request. Speed is an important factor, a user should not have to wait seconds or more to get a response.
What we accomplish with this information is to show the concrete role of the query distance parameter. The query distance, the time between each pull request from the core module to the connection module, currently is the most limiting factor, when it comes to throughput. This might chance, when certain module operations get more intensive and consuming.

In order to handle the n messages that shell be received during one authentication process, the pull timer has to fire at least n times. Therefore, the minimal time is $n * QUERY\_DISTANCE$. Limiting the amount of handled messages makes it possible to control the load the core is exposed to. Nevertheless, in general, handling multiple messages at once, is not a problem. If an request has not been finished before the next message arrives, they will be processed in parallel.

Our test program 'Speedtest', that can be found in the Java directory will measure the time that it takes from sending the first message till receiving the authentication response.
The aim of our test was to collect information about the correlation between query distance and the resulting request time. We took the average of 120 measurements for each query time configuration. Table 12 shows our measured results.

Table 12: Speed Test Results 1-50ms 120 request

Query Distance(ms)	min(ms)	avg(ms)	max(ms)
1	19,14	22,73	28,24
5	24,16	29,37	37,18
10	32,13	35,16	38,36
15	41,32	43,06	44,68
20	41,43	43,26	46,19
25	51,10	53,28	56,57
30	61,55	63,15	65,34
50	99,27	102,63	104,02

On a modern computer it took a request at least 19,14 milliseconds to be finished. This is therefore the minimal time a request needs to be handled by this prototype. We also observe that after setting this 20 milliseconds as our query distance, the resulting time is nearly double this time, as we expected.
The time itself is not that important, this will most certainly chance drastically, but the conclusion that can be derived from the measurement is: By knowing the minimal time that a requests needs to be finish, we can control how many authentication threads the core will spawn on average. A higher number will reduce CPU load, a lower will bring us near to the calculated minimal average. Furthermore, we may want to use this parameter to configure a static minimal response time.

11.3 Why Rust Instead of Java

Frequently the question 'why we used Rust instead of Java' comes up. Java is here used as place holder for all higher object oriented managed languages. The answer is distributed over the whole theses, at this point we want to summaries and add some aspects to the answer.

- The goal this of this work is to evaluate Rust as possible language for the agent. This of course does not nervelessly mean that Rust is the ideal choice on first sight. Nevertheless, evaluating that was the target.
- Rust was selected out of the domain of unmanaged languages where the resulting software can be build for a wide range of target platforms. Rust is an interesting language with currently rising interest by the community. It offers many promising features that were

partly introduced in Section 3. Many of the features fit to the requirements that such an security sensible software has.

- Finally, its the academic interest in this language. It allows the young object oriented programmer to experience an very pure object-functional system language that offers many known features from higher languages. Furthermore, it prevents typical mistakes that come with a system language, with built in code security checks.

11.4 Why Gaol instead of Docker

In this theses we introduce isolation and Gaol as our chosen framework to play around with it. This decision often raises the question, why we choose GOAL. We could have used Docker to isolate each of the modules and assign specific resources to each of these Docker instances. Using an solution like that would of course be more modern and would offer the benefit of a well tested heavenly used secure environment. Furthermore, future development could of choose this way. Nevertheless, our goal was something else and the following reasons are the basis for our decision to use Gaol.

- Gaol is an Rust implementation that uses known technologies like Namespaces and Chroot. When evaluating Rust as language for the Agent, it is in our opinion an important aspect to use these tools.
- The technologies used by Gaol are similar to the technologies used by docker. Inspecting and understand Gaol might be beneficial on an academic level.
- It is a rather small framework. Future developer may be able to learn from that example and extract what they need, to reduce the amount of dependencies.

12 Future Development

Implementing a prototype without an given budget or strict defined goals manifests to an never ending task. We had to stop somewhere and therefore tried to focus on a good demonstration. A lot of effort was put into a working prototype based on RPC communication integrated into a well designed project environment. On the way there trade-off where made. This section should summarize some of the ideas and plans, that we had but could not put into the current implementation.

12.1 Make it Panic Free

While inspecting the agent in Section 10.3.4 we saw that the agent will crash under many predictable error conditions. It is unavoidable to fix this issues by defining a recovery plan for situations where, for example a single module is not reachable or did not return in a given time. Therefore, it is required to extract critical program paths and handle them. In Rust,this is typically achieved by using the `result` enum. If the calling function receives an error, it should recover. The prototype often ignores such cases, gives an simple console output or panics.

12.2 TokioCore - Features

Currently each RPC call creates its own new 'Tokio Core Event Loop'. It is possible to spawn multiple features into one single event loop. To avoid the overhead that comes from creating this objects, a way should be developed to share a single loop or at least limit each connection to one single event loop. Before doing this, it will be required to study how many resource this overhead costs.

12.3 Fix Sandbox

As we mentioned already when taking about our Sandbox solution, the current way is more or less just an example that should show that it is possible to use a Rust sandbox and that it is working. We recall, that the problem we encountered is that it is not possible to allow an incoming connection.
The developer of Gaol also recommend a manager thread with whom the sandbox threads can talk via inter process calls. A solution like that may look similar to our current workaround, but may uses long running child threads, or external binaries and communicate with them via IPC instead of TCP.

12.4 Provide interface for Gaol

Currently the Gaol sandbox solution is used in the identity storage to limit the folders from which the storage can read. After finding a good and generic suitable design for the sandboxing it will be wise to extract this functionality into its own library and create an abstraction layer tailored to the personal agent. The privilege dropper was an optional part of the agent specification, but was let down because of the unsatisfying solution and the amount of time that was needed to come with something that was working in a way.
While the current solution is also limited to Linux, there are ways to create something similar for windows.

Using windows in a privilege safe way can be accived by using the securitybaseapi API. The functionalities that are most similar to what we did with GOAL are the AdjustTokenGroups function and the CreateRestrictedToken function. The AdjustTokenGroups function allows to disable groups that are available in the current access token. With the CreateRestrictedToken function one can create a new access token that is a restricted version of the existing token [22].
Alternatively, it would be possible to use Docker for each module or the whole agent. The container could easily be installed and used on one of multiple supported platforms. Within the container the known Linux restrictions could be used.

12.5 Sandbox in the Core

A way to simplify the usage and increase security would be to make the core, or an additional broker process responsible for the sandbox management. We discussed something similar in Section 2.2.7, with the Chrome sandbox solution. With that approach, this managing process could start and stop modules and place them in a separated individual customized sandbox. To implement this with Goal, a solution has to be found on how to start and stop loose processes.

12.6 Certificate based authentication for modules

Theoretically it is possible for an third party application to use the Rust library for our RPC interface definition, implement it and talk to our modules. We can use the TSL certificate to ensure only parties with the key and access to the certificate can use the RPC connections. That this is possible and how it works is shown for the connection between the authentication module and the core.

12.7 Protocol

Currently, the authentication does not follow any protocol or higher idea. Used data, how this data is handled, the format and way messages are sent, is all created to explore and show the possibilities a future developer has.
Therefore, one of the next steps should be to implement a strict and well design protocol, in which messages and data flows are defined. Also, the design and content of digital identities is just a demonstration and most likely not usable in the future. What should be usable is the expendable JSON library in which such things can be specified. Also the idea of fixed and flexibel parts in identities should be applicable on future scenarios.

12.8 Static Tests

Only a few static tests are currently implemented. Mainly, because at the moment testing would only verify demonstration functionalities. Nevertheless, in future versions all functions should be testable with static tests.
This may be of interest, as soon as an code and function specification is developed. Here it makes sense to create tests to develop forwards them or to write tests in a multi developer environment to verify already working solutions after code changes. Overall, tests a well integrated into the Rust environment.

12.9 Connection Module

The connection module is one of the first attack surfaces when attacks happen from the outside. At the moment, the module does not provide any real security. We could think of many good and important features that could be added to this module, the most crucial are: Blacklist for IP addresses that performed to many attempts, whitelist known scanner and receiver, and specify limits for the input data.

12.10 Authentication Module - Trusted Platform Module

A Trusted Platform Module is a piece of hardware the provides cryptographical relevant functionalities like key generation, random number generation, hash generation, encrypting and decrypting. Therefore, the Trusted Platform Module provides functions to sign and messages. However, this is hardware related and not strictly tied to a user, meaning it can be used for other applications to. Therefore, the Trusted Platform Module may also be used to provide a public key that authenticates the personal agent [19].

12.11 ISO24760

One step further into a trusted zone, would be to create an ISO certified solution. A major target for the agent is to provide a maximum on privacy. The standard defines some minimum capabilities for such an identity management system: Create support for minimal- selective- disclosure, minimize the ability to link identities, authenticate entities that use identity information, record and audit the use of identity information, support pseudonyms and some more [12, p. 17].

12.12 Identity Storage

Storing all digital identities on one device may lead to data lose. Losing this will be either a catastrophic event because all is gone, or the agent identities are additional stored on multiple 'public' places, where the agent can get this information back again. The second scenario makes the identities more vulnerable. Therefore, the agent might be the one entity that has access to all data, but for the storage a better solution should be created. Using a distributed storage like a Blockchain is an approach that is already in discussion and seems to be a good option. There are already multiple projects from which we introduced some, that implementing solution based on the idea of a distributed identity storage [14].

13 Conclusion

We discussed many digital identity related topics and highlighted the need of new independent solutions. The Digidow project could be such a solution. There are already multiple projects, which demonstrate the interest and need of a new, transparent, safe and broad usable way of identity management.
With the Personal Agent prototype we demonstrated multiple technologies that may be used for a future agent derived from an formal definition and following a well designed protocol. The resulting prototype is designed for a technology demonstration which combines multiple concepts and frameworks. Nevertheless, there is still a lot of research to perform in order to deploy this Rust prototype and realize a robust, secure, and usable web service for authentication.
While developing this prototype in Rust, we saw that isolation and standard cryptography, in our opinion two essential things, are possible and ready to use. Furthermore, we demonstrated and verified that these assets are working together with many others like the TLS secured RPC connections and basic communication between scanner and verifier.

A central target for this thesis was to create a personal agent in Rust to get to know and evaluate Rust as programming language. We conclude that Rust is a very well designed and strong system language. With some experience it is fun to use and enables us to rapidly create working system near prototypes. Overall, the resulting software turns out to be reliable and stable. Rust is already well integrated into some common developer tools and seems to stay alive for a few more years. The community is growing and it seems like they love Rust.
Is Rust the right choice for this project? A major drawback to consider is the complexity of this language, therefore, as always, it depends. Assuming that this agent will stay an university project, only a few developer will have the experience to produce reliable and good Rust code. If there are enough resources and know how for future Rust developers to engage with that language, it is definitely worth. Rust offers many features that fit well to the ideas of this project and helps to create a as secure as possible solution. On the other hand, we think that being confronted with an ongoing complex and sensitive Rust project, without experience, may either scare of some potential future Rust developer or reduce the quality. Nevertheless, as Rust enthusiasts and with a look on the academic benefits, we conclude with recommending Rust as language for this project.

Acknowledgement

I would first like to thank my thesis advisor Univ.-Prof. Priv.-Doz. Dr. René Mayrhofer of the Institute for Networks and Security at JKU Linz. He consistently allowed this paper to be my own work, but steered me in the right direction whenever I needed it.
I would also like to mention that something unique about Rust is the community. From my first step till the last step, I encounter numerous problems that I could not have solved myself. Every time the community stepped in and together we solved the problem (or file a bug report). Special thanks to *tikue*, a TarpC developer who helped my understanding the framework.
Finally, I must express my very profound gratitude to my partner and child for all their understanding for the countless hours I invested into this project. This accomplishment would not have been possible without them. Thank you.

References

[1] Jason Andress. *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice*. 1st ed. Syngress, 2011. ISBN: 1597496537,9781597496537.

[2] Blockstack. *Frequently Asked Questions*. Aug. 11, 2018. URL: https://blockstack.org/faq/.

[3] Bert Bates Bryan Basham Kathy Sierra. *Head First Servlets & JSP: Passing the Sun Certified Web Component Developer Exam*. 1st ed. SCWCD. O'Reilly Media, 2004. ISBN: 9780596005405,0596005407.

[4] Project Chromium. *Linux Sandboxing*. Ed. by Google. July 30, 2018. URL: https://chromium.googlesource.com/chromium/src/+/b4730a0c2773d8f6728946013eb812c6d3975bec/docs/linux_sandboxing.md.

[5] Project Chromium. *Sandbox*. Ed. by Google. July 30, 2018. URL: https://chromium.googlesource.com/chromium/src/+/b4730a0c2773d8f6728946013eb812c6d3975bec/docs/design/sandbox.md.

[6] DaGenix. *rust-crypto*. 2016. URL: https://github.com/DaGenix/rust-crypto (visited on 06/12/2018).

[7] Digidow. *Digidow*. Ed. by Institute of Networks and Security. Aug. 5, 2018. URL: https://ins.jku.at/research/projects/digidow.

[8] DockerTechnologie. *Docker*. Dec. 19, 2018. URL: https://docs.docker.com/engine/faq/#what-does-docker-technology-add-to-just-plain-lxc.

[9] Fido. *About The FIDO Alliance*. Aug. 16, 2018. URL: https://fidoalliance.org/about/overview/.

[10] Jens Getreu. *Embedded System Security with Rust: Case Study of Heartbleed*. July 30, 2018. URL: https://blog.getreu.net/projects/embedded-system-security-with-Rust/.

[11] ID2000. *ID2000*. Aug. 11, 2018. URL: https://id2020.org/.

[12] ISO. *ISO/IEC 24760-1:2011*. Tech. rep. defines terms for identity management, et al., 2011, p. 20. URL: https://www.iso.org/standard/57914.html.

[13] ITSEC. *Information Technology - Security Evaluation Criteria*. Tech. rep. Department of Trade and Industry, London, 1991.

[14] Ori Jacobovitz. *Blockchain for Identity Management*. Tech. rep. The Lynne and William Frankel Center for Computer Science Department of Computer Science, 2016.

[15] Michael Kerrisk. *Namespaces in operation*. Jan. 4, 2013. URL: https://lwn.net/Articles/531114/.

[16] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. 1st ed. No Starch Press, 2010. ISBN: 1593272200,9781593272203,159327291X,9781593272913.

[17] Arvind Kumar. *Rust is the Most Loved Programming Language in 2018*. Aug. 8, 2018. URL: https://www.technotification.com/2018/05/rust-most-loved-programming-language.html.

[18] Linux. *UNSHARE(2)*. 2018. URL: http://man7.org/linux/man-pages/man2/unshare.2.html (visited on 02/02/2018).

[19] P. Maene et al. 'Hardware-Based Trusted Computing Architectures for Isolation and Attestation'. In: *IEEE Transactions on Computers* 67.3 (Mar. 2018), pp. 361–374. ISSN: 0018-9340. DOI: 10.1109/TC.2017.2647955.

[20] Louise Matsakis and Issie Lapowsky. *Everything we know about Facebook's massive security breach*. Ed. by Wired. Sept. 28, 2018. URL: https://www.wired.com/story/facebook-security-breach-50-million-accounts/.

[21] Nicholas Matsakis and Aaron Turon. *Rust Lang Bool - 2018 Edition*. Jan. 1, 2018. URL: https://doc.rust-lang.org/book/2018-edition/.

[22] Microsoft. *Securitybaseapi*. 2019. URL: https://docs.microsoft.com/en-us/windows/desktop/api/securitybaseapi/.

[23] David Oberbeck. *FACHBEITRAG: Facebook Connect im Datenschutz-Check*. Aug. 10, 2018. URL: https://www.datenschutzkanzlei.de/fachbeitrag-facebook-connect-im-datenschutz-check/.

[24] Oswa. *Open Source Hardware Association*. Ed. by Michael. 2018. URL: https://www.oshwa.org/ (visited on 08/05/2018).

[25] Pat Patterson. *OpenID Connect: An Overview*. 2013. URL: https://de.slideshare.net/metadaddy/openid-connect-an-overview (visited on 08/10/2018).

[26] Persona.org. *How browsers interact with IdPs*. 2018. URL: https://archive.fo/20130129174959/http://identity.mozilla.com/post/7669886219/how-browserid-differs-from-openid (visited on 08/11/2018).

[27] Ring. *Ring*. 2018. URL: https://github.com/briansmith/ring (visited on 07/30/2018).

[28] Rustls. *Rustls*. 2018. URL: https://github.com/ctz/rustls (visited on 07/30/2018).

[29] Serde. *Serde*. 2018. URL: https://docs.serde.rs/serde/ (visited on 08/11/2018).

[30] Prabath Siriwardena. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Apress, 2014. ISBN: 978-1-4302-6818-5.

[31] Jon Skeet. *C-Sharp in Depth*. Manning, 2008.

[32] Sodiumoxide. *Sodiumoxide*. 2018. URL: https://github.com/dnaq/sodiumoxide (visited on 07/30/2018).

[33] TARPC. *TarpC*. Ed. by Google. 2018. URL: https://github.com/google/tarpc (visited on 06/12/2018).

[34] Rene Mayrhofer TEDxLinz. *Who are you... And how to prove your identity digitally?* 2017. URL: https://www.youtube.com/watch?v=hxnI553YDbQ (visited on 08/05/2018).

[35] R. Thurlow. *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC 5531. RFC, Aug. 25, 2018. URL: https://tools.ietf.org/html/rfc5531.

[36] TokioRS. *Tikio.rs*. 2018. URL: https://tokio.rs/docs/getting-started/hello-world/ (visited on 08/11/2018).

[37] UUID-RS. *UUID-RS*. 2018. URL: https://github.com/uuid-rs/uuid (visited on 08/11/2018).

[38] European Union. 'Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)'. In: *Official Journal of the European Union* L119 (May 2016), pp. 1–88. URL: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC.

[39] Munsif Vengattil and Peresh Dave. *Facebook now says data breach affected 29 million users, details impact*. Ed. by Reuters. Oct. 12, 2018. URL: https://www.reuters.com/article/us-facebook-cyber/facebook-now-says-data-breach-affected-29-million-users-details-impact-idUSKCN1MM297.

[40] WebPKI. *WebPKI*. 2018. URL: https://github.com/briansmith/webpki (visited on 07/30/2018).

[41] Phillip Windley. *Digital Identity*. Sebastopol: OReilly Media Inc, 2005. ISBN: 978-0-596-00878-9.

[42] WindowsHello. *Windows Hello for Business Overview*. 2017. URL: https://docs.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/hello-overview (visited on 08/16/2018).

[43] rust crypto. *rust-crypto*. 2018. URL: https://github.com/DaGenix/rust-crypto (visited on 07/30/2018).

[44] gaol. *gaol*. 2018. URL: https://github.com/servo/gaol (visited on 06/12/2018).

[45] serde js. *Serde JS*. Ed. by serde. 2018. URL: https://github.com/serde-rs/json (visited on 06/12/2018).

[46] killcup. *cargo-edit*. Aug. 11, 2018. URL: https://github.com/killercup/cargo-edit.