POLITECNICO DI TORINO

Master's Degree in Computer Engineering
(Cybersecurity)

Master's Degree Thesis

Ensuring integrity of MUD-enabled plug-ins for Smart Home Gateways

Supervisors

Prof. Fulvio CORNO

Dr. Luca MANNELLA

Candidate

Daniele DI BATTISTA

July 2023

Summary

Smart homes, equipped with various home automation systems, have gained popularity but also face significant cybersecurity challenges. To address these concerns, the Internet Engineering Task Force (IETF) introduced a new standard, Manufacturer Usage Description (MUD), which employs a white-list approach to enhance IoT security. This standard requires IoT device manufacturers to provide MUD files specifying the allowed communication endpoints, effectively mitigating the risk of unauthorized access and Distributed Denial of Service (DDoS) attacks.

One critical aspect of ensuring the security of smart home environments lies in the authentication of plug-ins and the associated MUD files. The authentication process plays a pivotal role in verifying the legitimacy of the plug-ins and the communication endpoints specified within the MUD files. However, this aspect is often overlooked or not given sufficient attention in existing research.

In order to enhance the authentication processes within smart home environments, the proposed solution will leverage the Codenotary Community Attestation service (CAS). CAS is a robust and reliable platform that provides attestation services for software artifacts, ensuring their integrity and authenticity.

By integrating CAS into the authentication framework, the master thesis aims to establish a trusted and verifiable chain of custody for plug-ins and their associated MUD files. CAS will generate cryptographic proofs, such as digital signatures to attest the authenticity and integrity of the submitted artifacts.

These cryptographic proofs will serve as evidence that the plug-ins and MUD files have not been tampered with or modified during transmission or storage. The smart home gateway can then verify the validity of these proofs. This process ensures that only authenticated and unaltered plugin-ins are accepted and processed by the gateway.

By leveraging CAS, the authentication framework adds an additional layer of trust and confidence to the authentication process. Plug-ins developers and smart home users can have increased assurance that the plug-ins and their associated MUD files are genuine, reducing the risk of unauthorized access or compromised security within the smart home environment.

By addressing the authentication challenges in the context of smart home gateways and MUD-enabled plug-ins, this research not only contributes to the overall security of smart homes but also establishes a foundation for trustworthy and secure interactions between different IoT devices and their associated plug-ins.

Acknowledgements

I take this opportunity to express my deepest gratitude to the individuals who have played a significant role in the completion of my Master's thesis. Their support has been instrumental in shaping this work.
First and foremost, I extend my heartfelt appreciation to Professor Fulvio Corno, who gives me the opportunity to participate in this work under his mentorship.
I like to extend my sincere thanks to Dr. Luca Mannella, my co-relator, for his valuable advice and for the time he dedicated to me.
I am grateful to my family. Their unconditional love, support and belief in my abilities have been a constant source of strength throughout my academic journey.
I am particularly grateful to my mother, Cinzia Persia, for her unwavering dedication in supporting me at every step of my studies. From assisting me with exams to helping me understand complex courses, her presence and guidance have been a source of reassurance and motivation.
Daniele Di Battista

Table of Contents

List of Tables

List of Figures

Glossary

Home Assistant

Home Assistant is an open-source home automation platform that allows users to control and automate various smart devices and services within their homes. It provides a centralized hub for managing and integrating different IoT devices, enabling users to create customized automation routines and control their smart home environment.

plug-ins

In Home Assistant, the term "plug-in" encompasses two distinct categories: integrations and add-ons. The focus of this thesis is specifically on integrations.

integrations

Integrations in Home Assistant refer to software components or plugins that facilitate seamless communication and control between Home Assistant and specific devices, services, or platforms. These integrations serve as bridges that enable Home Assistant to interact with and manage various devices and services within a smart home setup. Examples of integrations include lights, thermostats, door locks, cameras, sensors, and more.

HACS

HACS (Home Assistant Community Store) is a community-driven extension and plugin store for Home Assistant. It provides a platform for users to discover, install and manage custom components, integrations, themes and other community-contributed extensions to enhance the functionality and capabilities of Home Assistant.

MUD

MUD is a standardized framework that describes the expected network behavior and communication requirements of an Internet of Things (IoT) device. It defines the types of network connections an IoT device should establish and the associated security policies.

MUD Snippet

A MUD snippet is a small portion of the MUD file or description that contains specific rules and policies for network communication. It represents a subset of the complete MUD description and is typically associated with an individual IoT device or integration.

MUD Manager

MUD Manager is a software component or system responsible for managing the integration of MUD files or snippets into a network infrastructure. It handles tasks such as validating and applying MUD policies and ensuring the network's security and compliance.

Notarization

Notarization is the process of verifying and confirming the authenticity and integrity of software or digital artifacts. It involves obtaining a cryptographic signature or certificate from a trusted authority to ensure that the software or artifact has not been tampered with and can be trusted.

CAS

Codenotary CAS (Community Attestation Service) is a solution that provides cryptographic notarization services for software and code artifacts. It ensures the immutability and integrity of code by using cryptographic proofs, allowing organizations to verify the integrity of code and detect any unauthorized changes.

Chapter 1

Introduction

In today's rapidly advancing technological landscape, smart homes have become increasingly prevalent. These homes are equipped with various Internet of Things (IoT) devices that offer convenience and automation. From voice-activated assistants to automated lighting and temperature control, smart homes provide enhanced comfort and control for users. However, the widespread adoption of connected devices also brings significant security concerns.

Malicious attacks on smart home systems can compromise user privacy, disrupt daily routines and even pose physical risks. Intruders gaining unauthorized access to smart home devices can eavesdrop on conversations, manipulate security cameras or even tamper with critical systems such as locks and alarms. With the growing dependence on smart home technology, it is crucial to develop robust security mechanisms to protect these environments and ensure the safety of their occupants.

In the context of smart homes, the Home Assistant platform has emerged as a popular open-source home automation platform. It provides users with a centralized hub for controlling and monitoring various smart devices within their homes. Through the inclusion of integrations developed by a diverse communities of developers, Home Assistant offers an extensive range of features and functionalities, enabling users to customize and extend the capabilities of their smart home setups.

However, the flexibility and openness of the Home Assistant platform also introduce security challenges. One of the key concerns is the trustworthiness and integrity of the integrations included into the system. Third-party integrations, although providing additional functionality, can potentially introduce vulnerabilities or malicious code that compromise the security of the overall smart home system. Therefore, it is essential to develop an authentication and verification mechanism that ensures the trustworthiness and reliability of these integrations.

To address this challenge, the objective of this Master Thesis is to introduce an authentication and verification mechanism for the integrations in Home Assistant. The aim is to provide a reliable and secure approach to validate and verify the authenticity of integrations, allowing users to confidently incorporate third-party integrations into their smart home setups, knowing that they meet the required security standards.

To achieve this objective, this thesis builds upon a pre-existing work that explores the concept of Manufacturer Usage Description (MUD) as a promising approach in the field of smart home security. MUD enables devices to communicate their intended behavior and network access requirements, enabling network administrators to effectively enforce security policies. This pre-existing work has laid the foundation for enhancing the security of smart home systems by leveraging the insights and lessons learned from MUD.

Additionally, we investigate authentication mechanisms that can ensure the trustworthiness of integrations included into Home Assistant. One such mechanism we explore is the Codenotary Community Attestation Service (CAS). CAS is a trusted and reliable service that verifies the integrity and origin of digital assets, ensuring their trustworthiness and authenticity. By incorporating CAS into the authentication process, we can enhance the security and reliability of Home Assistant by ensuring that only validated and trusted integrations are utilized in the system.

In summary, this Master Thesis aims to enhance the security and reliability of smart home systems by introducing an authentication and verification mechanism for integrations in the Home Assistant platform in order to establish a trusted environment, enabling users to confidently incorporate third-party integrations into their smart home setups while maintaining the required security standards.

The remainder of this Thesis is organized as follows:

- **Chapter 2: Background** provides an overview of the relevant concepts and technologies in the field of smart home security, including a discussion on the Home Assistant platform, on the recent emergence of Manufacturer Usage Description (MUD) as a promising approach and existing solutions for authentication and verification mechanisms are also examined.

- **Chapter 3: Integrating MUD in Home Assistant** serves as the starting point for this thesis by presenting an extended MUD architecture specifically developed for Home Assistant. It provides a detailed exploration of the integration process, communication restrictions, and highlights the role of the MUD Generator integration.

- **Chapter 4: Authenticating integrations and MUD Snippets** delves into the core of the Thesis, presenting the authentication mechanism for integrations and MUD snippets in Home Assistant. It explores the integration notarization process, authenticity checks and introduces the use of the CodeNotary Community Attestation Service (CAS). CAS is a trusted and reliable service that verifies the integrity and origin of digital assets, ensuring their trustworthiness and authenticity. By incorporating CAS into the authentication process, the Thesis enhances the security and reliability of Home Assistant by ensuring that only validated and trusted integrations are utilized in the system.

- **Chapter 6: Conclusions** summarizes the contributions of this Thesis and outlines potential avenues for future research and improvement.

Chapter 2

Background

In this chapter we delve into the foundational aspects that form the basis of our research. We begin by exploring the concept of Home Assistant, a popular open-source platform that enables users to integrate and control smart devices within their homes. Next, we delve into the Manufacturer Usage Description (MUD), a standardized framework that aims to enhance the security of Internet of Things (IoT) devices by specifying their intended behavior and communication patterns. We then turn our attention to the state of the art authentication mechanisms employed in IoT systems, examining various approaches and highlighting their strengths and weaknesses. Lastly, we discuss the concept of notarization, an emerging technique that can be leveraged to ensure the integrity and authenticity of IoT device software. By comprehensively analyzing these topics, we lay the groundwork for our subsequent research and contribute to the broader understanding of secure and reliable IoT ecosystems.

2.1 Home Assistant

Home Assistant is an open-source home automation platform that offers users a comprehensive solution for controlling and monitoring smart devices within their homes. With its user-friendly interface and wide range of supported devices and protocols, Home Assistant has gained popularity among homeowners seeking to create a centralized automation system.

At its core, Home Assistant aims to simplify the management and integration of various smart home technologies. It provides a unified platform where users can configure and control their devices from a single interface, regardless of the brand or communication protocol. By bringing together devices such as lights, thermostats, sensors and security systems, Home Assistant empowers users to create personalized automations and routines. One of the key features of Home Assistant is its flexibility and extensibility. It supports a wide range of platforms, allowing users to connect and control devices from various ecosystems. Additionally, Home Assistant has a vibrant community that actively contributes to the development of new plug-ins and provides support to users. Specifically, Home Assistant encompasses two distinct “plug-in categories": *integrations* and *add-ons*. Both categories facilitate communication and control between Home Assistant and specific devices, services or platforms. While add-ons are basically Docker containers managed by Home Assistant, integrations are closer to the traditional plug-in concept — and for this reason they are the focus of this work.

2.1.1 Home Assistant architecture

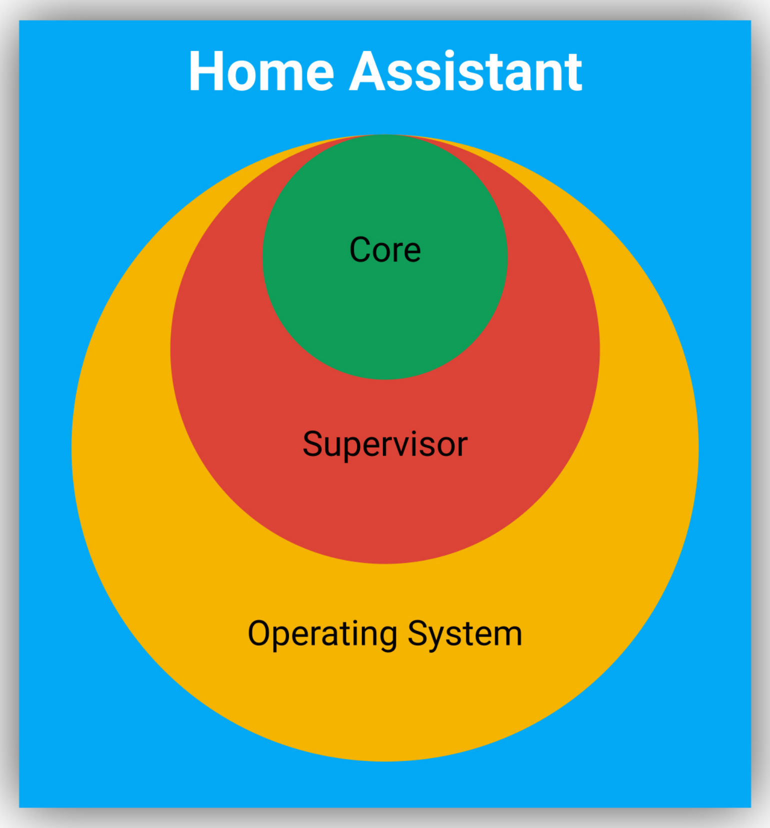


Figure 2.1: Architecture diagram of Home Assistant.

The architecture of Home Assistant is designed to provide a flexible and scalable framework for home automation. At its core, Home Assistant consists of several key components that work together to create a centralized and unified platform for controlling and managing smart devices.

The central component of Home Assistant is the Home Assistant Core. This component handles the core functionalities of the platform, such as processing automation rules, managing states and handling user interactions. It acts as the central hub for data flow, ensuring that information from various devices and integrations is properly processed and coordinated.

Another important component is the Home Assistant Supervisor. This layer serves as a management interface for Home Assistant, offering functionalities such as add-on management, system updates and advanced deployment options. The Supervisor ensures the reliability and stability of the platform, simplifying the installation and maintenance processes for users.

To provide a consistent and optimized environment for running Home Assistant, the Home Assistant Operating System (HassOS) is available for installation on supported devices. HassOS is a minimal, lightweight operating system that is specifically designed to run Home Assistant. It allows for efficient resource utilization and provides a reliable foundation for the platform.

Overall, the architecture of Home Assistant is designed to be modular and extensible, allowing users to tailor the platform to their specific needs.

2.1.2 Installation

Home Assistant offers four different installation methods. The last two installation methods are the ones recommended by the Home Assistant developers.

Home Assistant Core: This method entails a manual installation that utilizes a Python virtual environment. It offers advanced flexibility for expert users.

Home Assistant Container: This method allows for an independent installation of Home Assistant Core using containerization technology such as Docker.

Home Assistant Supervised: This method involves a manual installation of the Supervisor. It provides a greater level of control and customization.

Home Assistant Operating System: This method involves deploying a minimal operating system specifically optimized for powering Home Assistant. It includes the Supervisor, which efficiently manages Home Assistant Core and Add-ons.

The appropriate installation method can be selected based on the specific device and platform being utilized. This ensures compatibility and optimal performance for your Home Assistant setup.

2.1.3 Concepts and Terminology

To better understand Home Assistant, let's explore some key concepts and terminology:

- **Integration**: An integration is a software component that enables Home Assistant to connect and communicate with other software, platforms or hardware controllers. Integrations serve as bridges between Home Assistant and external systems, allowing seamless interoperability and control. For example, the Philips Hue integration enables Home Assistant to communicate with the Hue Bridge, which acts as the hardware controller for Philips Hue smart devices. By integrating the Hue Bridge, all connected and Home Assistant compatible devices associated with it, such as smart bulbs or sensors, can be discovered and managed within Home Assistant. Integrations provide specific functionality and APIs to establish the connection and facilitate the exchange of data, enabling users to monitor, control, and automate their devices and services through the unified Home Assistant interface.

- **Dashboard**: A dashboard in Home Assistant is a user interface that provides an overview of the smart home's status and allows users to interact with their devices. It typically displays relevant information, such as sensor readings, device states and active automations. Dashboards can be accessed via the Home Assistant web interface or through dedicated mobile apps.

- **Devices and Entities**: Devices and entities refer to physical or virtual objects within the smart home environment that can be controlled or monitored. A device represents a specific physical device, such as a smart bulb or a smart lock. An entity, on the other hand, represents a specific aspect or attribute of a device, such as the brightness of a bulb or the locked status of a lock. Entities allow fine-grained control and automation of individual device features.

- **Automations**: Automations in Home Assistant are rules or routines that define actions to be performed automatically based on specific triggers or conditions. For example, an automation can be set up to turn on the lights when motion is detected or to adjust the thermostat based on the time of day. Automations enable the creation of personalized and intelligent behaviors within the smart home.

- **Scenes**: Scenes in Home Assistant allow users to capture and replicate specific states of multiple devices or entities. By defining a scene, users can easily restore a desired combination of device settings or create ambiance with a single command or trigger. For instance, a "Movie Night" scene can dim the lights, lower the blinds and turn on the TV.

- **Scripts**: Scripts in Home Assistant are sequences of predefined actions that can be executed on demand or as part of an automation. Scripts allow users to create custom sequences of commands or interactions with multiple devices or services. They offer flexibility and customization for specific scenarios or user preferences.
- **Add-ons**: Add-ons are optional packages or extensions that can be installed in Home Assistant to enhance its functionality. Add-ons can provide additional integrations, services or tools to expand the capabilities of Home Assistant. For example, there are add-ons for data analytics, voice assistants or advanced automation engines.

2.1.4 Security Considerations

Ensuring the security of Home Assistant and the connected IoT devices is crucial for maintaining a safe and protected smart home environment. In this subsection, we will explore important security considerations related to Home Assistant.

By addressing these security considerations and adopting security best practices, Home Assistant can provide a secure foundation for managing and controlling the IoT devices within a smart home environment.

Secure Communication Protocols

Home Assistant supports various communication protocols to interact with IoT devices and services. It is essential to prioritize secure protocols, such as Hypertext Transfer Protocol Secure (HTTPS) [1], Message Queues Telemetry Transport (MQTT) [2] with Transport Layer Security (TLS) [3], or Z-Wave Security [4], for encrypted and authenticated communication. By utilizing secure protocols, sensitive data transmission can be safeguarded from eavesdropping and tampering.

Authentication and Authorization

Strong authentication mechanisms are vital to prevent unauthorized access to Home Assistant. It is recommended to enable two-factor authentication (2FA) to add an extra layer of security to user accounts. Additionally, Home Assistant provides granular access control through user roles and permissions. Administrators should carefully manage user accounts and assign appropriate permissions to ensure that only authorized individuals can access and modify the system.

Regular Updates and Patching

To mitigate security vulnerabilities, it is essential to keep Home Assistant and its dependencies up to date by applying regular updates and patches. Home Assistant provides automatic update mechanisms, which should be enabled to ensure that the latest security fixes and enhancements are promptly installed. Regularly reviewing the release notes and security advisories is recommended to stay informed about any vulnerabilities or updates affecting the system. However, it is equally important to verify that these updates do not conflict with or compromise the functionality of components developed by individual developers. To maintain stability and avoid potential disruptions, developers are encouraged to rigorously test updates in a controlled environment that mimics their production setup. This verification process allows for early detection of any issues or incompatibilities that may arise due to the updates.

Network Segmentation

Segregating the smart home network can enhance security. Consider creating separate network segments or VLANs for IoT devices, isolating them from critical network infrastructure.

Third-Party Integrations and Add-Ons

When utilizing third-party integrations or add-ons, exercise caution and ensure they come from trusted sources. Verify that the integrations have undergone security assessments, have a reliable update process and have a strong community following. Regularly review and update third-party integrations and add-ons to benefit from the latest security enhancements.

2.1.5 Community and Ecosystem

Home Assistant benefits from a vibrant community that actively contributes to its development and support. The community-driven nature of Home Assistant has resulted in a wide range of add-ons, integrations and custom components. These community contributions expand the platform's capabilities, allowing users to enhance their smart home experience and address specific security requirements.

Furthermore, the Home Assistant community emphasizes security and actively collaborates to improve the platform's security posture. Regular security audits, bug bounty programs and continuous development efforts ensure that Home Assistant evolves to meet the growing security challenges in the IoT landscape.

2.1.6 Use Cases and Real-World Examples

Home Assistant offers a wide range of use cases and has been adopted in various real-world scenarios. In this subsection, we will explore some common use cases and provide real-world examples of how Home Assistant is being utilized.

1. Smart Home Automation

Home Assistant excels in creating a comprehensive and automated smart home experience. Users can integrate different smart devices, such as lights, thermostats, sensors and cameras into Home Assistant. This allows for centralized control, scheduling and automation of various home functions. For example, Home Assistant can automatically adjust the lighting based on occupancy, regulate the thermostat based on weather conditions or send notifications when unexpected events occur.

2. Energy Management and Efficiency

Home Assistant provides tools and integrations to monitor and manage energy consumption in a smart home. By connecting to smart meters, energy monitors or smart plugs, users can track energy usage, identify energy-hungry devices and implement energy-saving strategies. Home Assistant can analyze energy data, provide insights and trigger actions to optimize energy efficiency, such as automatically turning off standby devices or adjusting thermostat settings.

3. Security and Surveillance

Home Assistant can serve as a robust security and surveillance system. By integrating security cameras, motion sensors, door/window sensors and alarm systems, users can monitor their home's security status in real-time. Home Assistant can send instant alerts and notifications when security events occur, allowing users to take immediate action. Additionally, Home Assistant's automation capabilities can simulate occupancy by controlling lights and media devices, enhancing home security when residents are away.

4. Environmental Monitoring and Control

Home Assistant supports various environmental sensors and weather services, enabling users to monitor and control environmental conditions. For example, users can integrate air quality sensors, humidity sensors or weather services to receive real-time data and take appropriate actions. Home Assistant can automatically adjust ventilation systems, notify users of poor air quality or activate irrigation systems based on weather forecasts, ensuring a comfortable and healthy living environment.

Real-World Examples

Numerous users have deployed Home Assistant in their smart homes, showcasing its versatility and effectiveness. Some real-world examples include:

- **Ubiquiti**: Ubiquiti, a renowned networking and surveillance solutions provider has integrated Home Assistant into their UniFi Protect ecosystem[5]. This integration allows users to manage and monitor their Ubiquiti surveillance cameras, access points and network devices through Home Assistant's unified interface.
- **IKEA**: IKEA, the multinational home furnishings retailer has partnered with Home Assistant to enable seamless integration with their smart home products. This integration allows users to control and automate IKEA's TRÅDFRI[6] line of smart lighting products, including bulbs, controllers and motion sensors through Home Assistant.
- **Google**: Home Assistant has integration with Google Assistant[7], allowing users to control their Home Assistant-powered smart home devices using voice commands through Google Home devices or the Google Assistant app.
- **Philips**: Home Assistant supports integration with Philips Hue[8] smart lighting products. Users can connect their Hue lights to Home Assistant and have full control over their lighting, including adjusting colors, brightness and creating custom lighting automations.

These real-world examples demonstrate the versatility and practicality of Home Assistant in various contexts. Home Assistant's flexibility and extensive integrations enable users to customize and tailor their smart home experience to their specific needs.

2.1.7 Home Assistant Community Store (HACS)

The Home Assistant Community Store, commonly referred to as HACS, is an integral part of the Home Assistant ecosystem. It is a user-driven repository of custom integrations and add-ons created by the Home Assistant community. In this subchapter, we will explore the features and benefits of the Home Assistant Community Store.

1. Community-Driven Contributions

The Home Assistant Community Store serves as an invaluable addition to the Home Assistant ecosystem, complementing its inherent expandability through community-driven contributions. While Home Assistant itself offers the ability for users to extend its functionality, the community store provides distinct advantages and differences compared to submitting a integration into the regular ecosystem.

Firstly, the community store offers a dedicated platform specifically designed for users to share their custom integrations and add-ons with others. It serves as a centralized hub, consolidating the collective knowledge and expertise of the vibrant Home Assistant community. By leveraging the community store, users gain access to a wide set of extensions created and maintained by community members. This diversity enriches the ecosystem, offering users an extensive range of options to enhance their Home Assistant setup.

Furthermore, the community store facilitates seamless discovery and installation of community-developed extensions. It simplifies the process for users to explore, evaluate, and integrate new functionalities into their Home Assistant environment. Through the store's user-friendly interface and efficient installation mechanisms, users can effortlessly enhance their smart home setup with the contributions of fellow community members.

Additionally, the community store fosters collaboration and encourages active participation within the Home Assistant community. By providing a platform for developers to showcase their creations, it promotes engagement and knowledge-sharing among community members. Users can benefit from the collective expertise and continuous development efforts, as well as receive support and feedback from the community.

2. Extended Functionality

By utilizing the Home Assistant Community Store, users can extend the capabilities of their Home Assistant installation. They can discover and install custom integrations that are not available in the official Home Assistant repositories. These custom components enable integration with specific devices, services or platforms that are not natively supported by Home Assistant, broadening the range of compatible devices and expanding the integration possibilities.

3. User-Friendly Interface

The Home Assistant Community Store provides a user-friendly interface within the Home Assistant web interface. Users can browse through the available integrations and add-ons, read descriptions, view ratings and access documentation. The interface also allows for easy installation, configuration and updates, providing a seamless and streamlined experience for users.

4. Community Support and Collaboration

The Home Assistant Community Store fosters a spirit of collaboration and support within the Home Assistant community. Users can actively engage with the developers and other community members through forums, chat platforms and GitHub repositories. This open and collaborative environment promotes knowledge sharing, troubleshooting and the continuous improvement of community-driven extensions.

5. Customization and Personalization

With the Home Assistant Community Store, users have the freedom to customize and personalize their Home Assistant installation according to their preferences and requirements. They can choose from a vast array of community-developed themes, frontend customizations and additional functionality, allowing them to create a unique and tailored smart home experience.

While the Home Assistant Community Store offers numerous advantages, it is important to acknowledge that integrations available in HACS are generally less controlled compared to the official integrations present in the Home Assistant release. Since HACS relies on community contributions, there may be variations in the quality, reliability and compatibility of the available integrations. This lack of control has inspired the objective of my thesis that is to implement an authentication and verification mechanism for community developed integrations. By introducing such a mechanism, it aims to enhance the security and reliability of the community-driven integrations, providing users with a more trusted and curated selection of integrations. This authentication and verification mechanism will enable users to have greater confidence when selecting and installing integrations from the community store, mitigating potential risks associated with unverified or malicious ones.

In conclusion, the Home Assistant Community Store is a valuable resource that empowers users to enhance and customize their Home Assistant installations. It fosters community collaboration, provides extended functionality through custom integrations and add-ons and offers a user-friendly interface for easy discovery, installation and updates.

2.2 The Manufacturer Usage Description

The Manufacturer Usage Description (MUD) is a component-based architecture that plays a crucial role in enhancing the security and access control of Internet of Things (IoT) devices. MUD provides a standardized mechanism for end devices to communicate their access and network requirements to the network infrastructure. By doing so, MUD enables network administrators to enforce device-specific access policies and improve the overall security posture of the IoT ecosystem.

The goal of MUD, as specified in RFC 8520 [9], is to enable end devices to signal to the network what sort of access and network functionality they require to properly function. Initially focused on access control, MUD offers a versatile framework that can be extended to cover other aspects as well. RFC 8520 defines the MUD architecture and specifies the necessary components, including MUD files, network enforcement points and MUD managers.

MUD leverages standardized protocols and formats, such as Yet Another Next Generation (YANG) [10] modules, Dynamic Host Configuration Protocol (DHCP) [11] options, Link Layer Discovery Protocol (LLDP) [12], Uniform Resource Locators (URLs) [13], X.509 [14] certificate extensions and signing mechanisms, to facilitate the exchange of device requirements and enable seamless integration with network infrastructure. By employing MUD, network administrators gain granular control over device access permissions, reducing the attack surface and minimizing the risk of unauthorized access.

2.2.1 MUD Architecture and Components

The Manufacturer Usage Description (MUD) architecture, as defined in RFC 8520 [9], encompasses several key components that work together to enable the enforcement of device-specific access policies. These components play a vital role in ensuring that IoT devices can effectively and securely communicate their network requirements.

The main components of the MUD architecture, as depicted in the RFC 8520 diagram, include:

- **MUD Files**: MUD files are JSON-based documents that contain detailed descriptions of the behavior, capabilities and network requirements of IoT devices. Device manufacturers provide these files specifying the access policies that should be enforced for each device. MUD files follow a standardized format and include information such as device classes, required network protocols and allowed network services.

- **MUD Manager**: The MUD Manager acts as the central coordinating entity in the MUD ecosystem. It manages the distribution and storage of MUD files, ensuring their availability to network enforcement points. The MUD Manager interacts with device manufacturers to retrieve MUD files and maintains a repository of these files. It facilitates secure and efficient MUD file exchange between manufacturers, network administrators and other MUD components.

- **MUD File Server**: The MUD File server is responsible for hosting and serving MUD files. It provides a reliable and accessible location where MUD files can be retrieved by network enforcement points. The MUD File server, often maintained by the manufacturer or a trusted third party, ensures the integrity and availability of MUD files.

- **Policy Enforcement Point**: Policy Enforcement Points (PEPs), represented by routers or switches in the MUD architecture, play a crucial role in enforcing access policies based on the information provided in MUD files. As stated in the RFC 2753 [15], a PEP is the point where the policy decisions are actually enforced. PEPs examine network traffic and make access control decisions to ensure that IoT devices have appropriate network access. By referring to MUD files, PEPs enforce device-specific access policies and prevent unauthorized or malicious activities.

- **Thing**: The Thing refers to the IoT devices themselves, such as sensors, actuators or smart appliances. These devices connect to the network and communicate their network requirements using MUD. By incorporating MUD support, IoT devices can convey their intended behavior, required network services and access constraints to the network infrastructure.

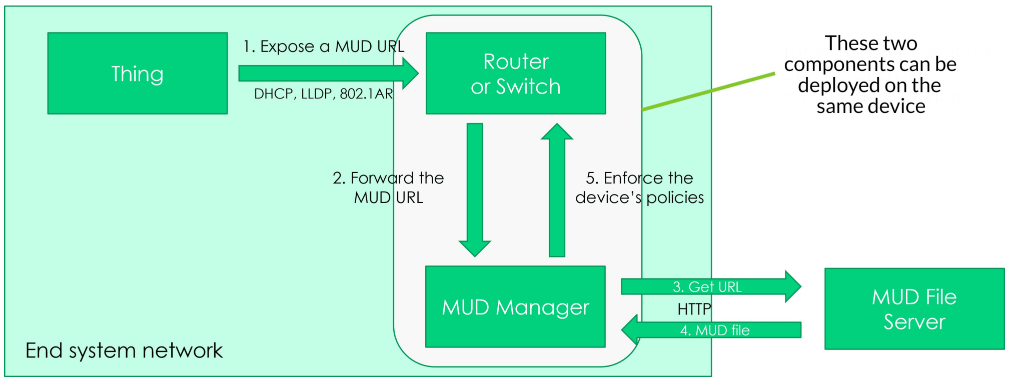


Figure 2.2: Architecture diagram of MUD.

In the context of Home Assistant, the integration of MUD brings significant benefits. It allows for improved security by defining and enforcing access policies tailored to individual devices, mitigating the risks associated with untrusted or compromised devices. Furthermore, MUD simplifies device management by providing automated configuration and identification, reducing the manual effort required for secure device onboarding. By leveraging these components, the MUD architecture provides a standardized framework for device manufacturers, network administrators and home automation systems to collaborate effectively in enforcing access control policies.

2.2.2 Structure of a MUD file

The MUD file is a critical component of the MUD framework and serves as the authoritative source for describing a device's behavior within a network. It contains the necessary information for smart home gateways to make informed decisions about network access controls and security policies. The MUD file adheres to a specific structure and includes various sections, each capturing essential aspects of the device's behavior.

The sections typically found in a MUD file include:

1. **Manufacturer Information:** This section provides details about the device manufacturer or vendor, such as the manufacturer name, the manufacturer's domain and contact information.

2. **Device Information:** This section contains specific details about the device itself, including the device name, model name or identifier and firmware version.

3. **MUD URL:** The MUD URL section specifies the location from which the MUD file can be retrieved. It includes the scheme (e.g., HTTP or HTTPS), authority (e.g., domain name or IP address), path and optional query parameters.

4. **MUD Rules:** This section defines the network communication and behavior of the device. It includes Access Control Lists (ACLs) which specify the permitted inbound and outbound network traffic and may incorporate information about local network traffic, external network traffic, supported protocols, port ranges and any URLs associated with the device.

The content within the MUD file is typically written using a specialized syntax defined by the MUD specification. This syntax enables the concise and standardized representation of the device's behavior and requirements.

By examining the components and sections within the MUD file, smart home gateways can effectively enforce network security policies, allowing devices to operate securely within the smart home ecosystem.

The following is an example of a possible MUD file. This example contains two access lists that are intended to provide outbound access to a cloud service on TCP port 443.

In this example, two policies are declared: one from the Thing and the other to the Thing. Each policy names an access list that applies to the Thing and one that applies from the Thing. Within each access list, access is permitted to packets flowing to or from the Thing that can be mapped to the domain name of `service.bms.example.com`. For each access list, the enforcement point should expect that the Thing initiated the connection.

Listing 2.1: MUD Snippet example

```
{
  "ietf-mud:mud": {
  "mud-version": 1,
    "mud-url": "https://lighting.example.com/lightbulb2000",
    "last-update": "2019-01-28T11:20:51+01:00",
     "cache-validity": 48,
     "is-supported": true,
     "systeminfo": "The BMS Example Light Bulb",
     "from-device-policy": {
       "access-lists": {
         "access-list": [
            {
              "name": "mud-76100-V6fr"
            }
          ]
        }
      },
      "to-device-policy": {
        "access-lists": {
          "access-list": [
            {
              "name": "mud-76100-v6to"
            }
          ]
        }
      }
    },
    "ietf-access-control-list:acls": {
      "acl": [
        {
          "name": "mud-76100-v6to",
          "type": "ipv6-acl-type",
          "aces": {
            "ace": [
              {
                "name": "c10-todev",
                "matches": {
                  "ipv6": {
                    "ietf-acldns:src-dnsname": "text.example.com",
                    "protocol": 6
                  },
                  "tcp": {
                    "ietf-mud:direction-initiated": "from-device",
                    "source-port": {
                      "operator": "eq",
                      "port": 443
                    }
                  }
                },
                "actions": {
                  "forwarding": "accept"
                 }
                }
              ]
            }
          }
        ]
      }
}
```

2.2.3 Security Considerations

In this subsection, we will discuss security considerations related to the deployment of the Manufacturer Usage Description (MUD) framework as outlined in RFC 8520.

Trustworthiness of MUD Files

To support the security of the MUD ecosystem, robust measures are in place to ensure the trustworthiness of MUD files. As stated in the RFC 8520, manufacturers and distributors employ secure mechanisms to sign MUD files, guaranteeing the integrity and authenticity of these files. By utilizing cryptographic signatures, devices and MUD managers can verify the legitimacy of MUD files before enforcing access control policies. Furthermore, the transmission of MUD files occurs through secure channels, predominantly using HTTPS, to safeguard against unauthorized modifications.

Mitigating MUD-Related Attacks

The adoption of MUD introduces potential attack surfaces and associated vulnerabilities. However, effective countermeasures have been implemented to mitigate MUD-related attacks. These measures include protecting against forged or malicious MUD files through the implementation of cryptographic signatures. Secure mechanisms are established for retrieving MUD files, employing encrypted communication channels such as HTTPS. Additionally, continuous monitoring and analysis of MUD-enforced access control decisions are conducted to detect and respond to any anomalous behavior promptly. These measures collectively contribute to maintaining a robust security posture within the MUD ecosystem.

Privacy Considerations

MUD deployment may involve sharing device-specific information with the MUD manager for access control enforcement. Privacy considerations should include defining appropriate data protection measures, ensuring compliance with relevant privacy regulations and minimizing the collection and storage of sensitive information.

Operational Security

Maintaining the operational security of MUD components is crucial. Considerations should include secure management of MUD managers, protection against unauthorized access or tampering with MUD-related configurations and monitoring for potential security incidents or anomalies in MUD enforcement.

Transmission of MUD URLs

The transmission of MUD URLs presents a critical point in the security of the MUD framework. When using protocols like DHCP[11] or LLDP[12], the transmission of MUD URLs remains vulnerable to potential security risks.

In the case of DHCP, where devices automatically obtain network configurations, including MUD URLs, the transmission of this information can be intercepted or manipulated by malicious actors. Unauthorized modification of the MUD URL could lead to devices receiving incorrect or malicious MUD files compromising the integrity and security of the access control enforcement.

Similarly, when utilizing LLDP for MUD URL transmission, the information can be exposed to eavesdropping or tampering during network discovery. Malicious actors can intercept or modify the MUD URLs, potentially leading to unauthorized or compromised access control policies.

To mitigate these vulnerabilities, it is crucial to employ additional security measures when transmitting MUD URLs. IEEE proposed to use 802.1AR[16] protocol, which, accoring to RFC 8520[9], provides a certificate-based approach to communicate device characteristics. Implementing secure communication channels such as HTTPS for the retrieval of MUD files further enhances the security posture and safeguards against unauthorized modifications.

By addressing the vulnerability in the transmission of MUD URLs, the overall security of the MUD ecosystem can be strengthened, ensuring the reliable and secure enforcement of access control policies for IoT devices.

2.3 State of the Art Authentication Mechanisms

Authentication mechanisms play a crucial role in ensuring data integrity, source verification and protection against data poisoning attacks. In this section, we will briefly explore different authentication solutions adopted by well-known companies and propose the adoption of Codenotary Community Attestation Service (CAS) for its unique features and advantages.

2.3.1 Existing Authentication Solutions

Android Ecosystem

The Play Integrity Application Programming Interface (API) authentication mechanism plays a crucial role in ensuring the security and integrity of data transmitted between applications and the Google Play Store. This authentication process is designed to verify the authenticity of requests made to the Play Integrity API, thus preventing unauthorized access and protecting against potential fraudulent activities.

At its core, the Play Integrity API authentication mechanism utilizes a combination of cryptographic techniques and secure communication protocols to establish a secure connection between the client application and the Google Play Store servers. The primary goal is to validate the identity of the client application and ensure that the data exchanged during the communication remains confidential and tamper-proof.

To initiate the authentication process, the client application needs to obtain an API key from the Google Play Developer Console. This API key serves as a unique identifier for the application and acts as a secret token that authenticates the requests sent to the Play Integrity API. The API key must be securely stored within the client application, as its compromise could lead to unauthorized access or misuse.

Once the client application has acquired the API key, it includes this key in the requests it sends to the Play Integrity API. These requests are typically made using secure communication protocols such as HTTPS to encrypt the data and establish a secure channel between the client and the server.

Upon receiving a request, the Play Integrity API verifies the authenticity of the client application by validating the API key included in the request. This validation process involves cryptographic algorithms and server-side checks to ensure that the provided API key matches the one associated with the client application.

If the API key is successfully validated, the Play Integrity API proceeds to process the request and perform the requested integrity checks on the data provided by the client application. These integrity checks may include verifying the digital signatures of the application packages, validating the Android Application Package (APK) files and ensuring that the application adheres to Google Play policies and guidelines.

In the event of a failed authentication attempt, the Play Integrity API rejects the request indicating that the client application's identity could not be verified. This rejection helps protect against malicious or unauthorized applications attempting to manipulate or exploit the Play Store's integrity system.

Overall, the Play Integrity API authentication mechanism is a vital component in maintaining the security and integrity of the Google Play ecosystem. By securely authenticating client applications and validating the data they provide, this mechanism helps ensure that only legitimate and trusted applications are available to users, fostering a safer and more reliable app ecosystem.

Apple Ecosystem

The Apple ecosystem incorporates a robust set of security mechanisms to ensure the integrity and trustworthiness of code and applications running on their devices. By employing a combination of system security, app security and certification processes, Apple strives to provide a secure environment for its users.

A fundamental component of Apple's security architecture is the secure boot chain. During the startup process, each stage of the boot chain is cryptographically signed, ensuring that only trusted code is executed. This chain of trust begins with the device's hardware and firmware, progressing through each layer of the operating system until the user interface is loaded. By verifying the integrity of software components at startup, Apple mitigates the risk of unauthorized modifications and safeguards against malicious code execution.

To further support security, Apple employs a multi-layered approach when it comes to app security. Before an application can run on Apple devices, it undergoes a thorough review and approval process. This process involves stringent guidelines and checks to identify any potential security vulnerabilities or violations. By scrutinizing apps prior to their inclusion in the App Store, Apple aims to protect users from harmful or malicious software.

In addition to the app review process, Apple's notarization system plays a crucial role in enhancing security for macOS users. Notarization is a process where developers submit their apps to Apple for a comprehensive security scan. This scan examines the app for any malicious code, ensuring that it adheres to Apple's security standards. Notarized apps receive a digital signature from Apple, indicating that they have undergone this verification process, thereby boosting user confidence in the app's integrity.

Moreover, Apple's ecosystem includes built-in antivirus protection for macOS. This antivirus feature constantly monitors and scans files on the system, including those downloaded from the internet. It actively detects and neutralizes known malware threats reducing the risk of infection and enhancing overall system security.

By combining secure boot chain mechanisms, app security reviews, notarization processes and built-in antivirus protection, Apple has created a comprehensive security framework for its ecosystem. These measures work in unison to ensure that only trusted code and applications are allowed to run on Apple devices. As a result, users can confidently download and use apps from the App Store or other trusted sources, knowing that their devices are protected against potential security risks.

Microsoft Ecosystem

Within the Microsoft ecosystem, a range of security measures are employed to prioritize publisher verification and trust validation, ensuring the integrity and safety of software, including macros, ActiveX controls and add-ins. Microsoft places emphasis on enabling users to verify the identity and credentials of publishers before trusting their code, mitigating potential security risks.

One key aspect of Microsoft's approach to security is the emphasis on publisher verification. Users are encouraged to exercise caution and diligence when interacting with software components such as macros, ActiveX controls or add-ins. Before executing or enabling these components, users are advised to verify the identity and credibility of the publisher. This verification process helps establish trust in the origin and integrity of the code, reducing the likelihood of malicious or unauthorized activity.

To further reinforce security, Microsoft has implemented the Microsoft 365 App Compliance Program, which offers a layered approach to app security and certification. This program aims to enhance user confidence in the applications utilized within organizations. Through the program, Microsoft provides a comprehensive set of security checks and validation processes for apps seeking certification. These checks encompass various aspects, including data handling, privacy compliance, security protocols, and adherence to industry best practices. By undergoing this rigorous certification process, apps can demonstrate their commitment to security and establish themselves as trusted solutions within the Microsoft ecosystem.

Microsoft's multifaceted approach to security not only emphasizes publisher verification but also fosters a proactive stance in promoting secure software practices. By encouraging users to verify publishers and establishing a comprehensive app compliance program, Microsoft prioritizes the protection of user data and system integrity. These measures collectively contribute to creating a secure ecosystem that empowers users to make informed decisions about the software they trust and use, thereby mitigating potential security risks.

2.3.2 Codenotary Community Attestation Service (CAS)

CAS is a notable example of notarization service. Notarization is a process that provides integrity and authenticity assurance for digital assets. It involves creating a tamper-evident record, commonly referred to as a notarized proof that can be used to verify the integrity and origin of the asset at a later time.

It utilizes blockchain principles with the underlying technology of immudb[17] to create an immutable and transparent record of the digital asset.

Blockchain[18] is a decentralized and distributed digital ledger technology[19] that records transactions or data across multiple computers. It operates on a peer-to-peer network where each participant maintains a copy of the entire blockchain.

This decentralized nature ensures transparency, immutability and trust in the recorded data.

Immudb is an open-source immutable database built using blockchain principles. It utilizes cryptographic hashing and merkle tree structures (a Merkle tree structure is a hierarchical data structure used in cryptography and computer science to efficiently verify the integrity and authenticity of data) to ensure data integrity and tamper-evidence. Immudb's design guarantees that once data is recorded, it cannot be modified without detection. This makes immudb an ideal solution for applications that require verifiable and tamper-proof data storage.

CAS Overview

CAS offers a comprehensive solution to give digital assets a meaningful, globally-unique and immutable identity. It ensures authenticity, verifiability and traceability from anywhere. It provides the ability to attest digital assets, adding a chosen trust level, custom attributes and meaningful status.

The key features and benefits of Codenotary CAS for notarization include:

- **Notarization Process:** CAS enables notarization by calculating the Secure Hash Algorithm 256-bit (SHA-256) [20] hash of digital assets and cryptographically signing the hash, status and trust level. This process creates a tamper-proof record that binds the asset's information together.

- **Immutability and Tamper-Evidence:** By leveraging immudb's blockchain-based technology, Codenotary CAS achieves immutability and tamper-evidence. Once a asset's hash or fingerprint is recorded in immudb, it becomes immutable and resistant to tampering. Any attempts to modify or tamper with the asset will be immediately detected, ensuring the integrity and authenticity of the notarized asset.

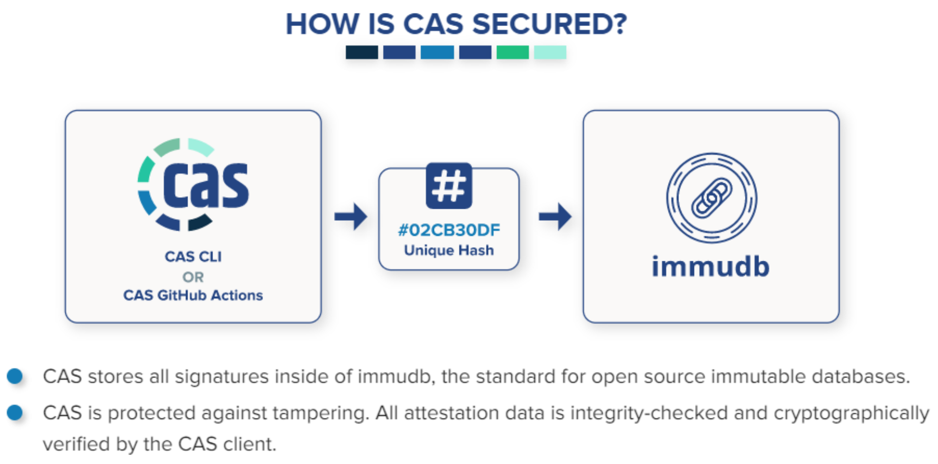
- **Decentralization and Trustworthiness:** The decentralized nature of immudb's distributed network of nodes adds an additional layer of trust and reliability to Codenotary CAS. Unlike centralized systems, which rely on a single authority, the decentralized nature of immudb ensures that verification is not dependent on a single point of failure. This decentralization enhances trustworthiness by eliminating the risk of manipulation or compromise by a single entity.

- **Verifiability:** The integrity and authenticity of the notarized proofs can be easily verified by comparing them against the original digital assets. This transparency allows developers and code consumers to independently verify the authenticity and integrity of the notarized assets.

- **Efficient and Lightweight Storage:** Immudb, the underlying technology used by Codenotary CAS, is designed to be lightweight and efficient. It provides a scalable storage solution that can handle large volumes of hashes without compromising performance. This efficiency ensures a seamless notarization process, enabling developers to quickly and reliably notarize their assets while maintaining optimal system performance.

- **Flexibility and Integration:** CAS can be seamlessly integrated into existing workflows and systems. It provides developers with a Command-Line Interface (CLI) for easy integration but unfortunately it doesn't offer APIs for programmatic access.

- **Open-Source and Community Support:** Both Codenotary CAS and Immudb are open-source projects, fostering collaboration and community support. The open-source nature allows developers to contribute, audit the code, and participate in the evolution of the technology. This collaborative approach ensures continuous improvements, security enhancements and innovation in the notarization and blockchain space.



- CAS stores all signatures inside of immudb, the standard for open source immutable databases.
- CAS is protected against tampering. All attestation data is integrity-checked and cryptographically verified by the CAS client.

Figure 2.3: How CAS is secured

By leveraging Codenotary CAS, we can address the challenges of data poisoning, data integrity and source verification effectively. Its robust features and integration capabilities make it a suitable choice for ensuring the authenticity and integrity of digital assets in our environment.

In the next subsection, we will delve into the technical implementation details and further explore the benefits and potential use cases of Codenotary CAS Service.

2.3.3 Notarization and Verification process using CAS Executable

In this subsection, we will describe the process of notarizing and verifying a digital asset using the Codenotary CAS executable. Codenotary CAS is a decentralized, tamper-evident notarization service that utilizes blockchain technology. Its purpose is to ensure the integrity and immutability of digital assets, enabling secure verification.

The notarization process involves the following steps:

Preparation

Before notarizing a digital asset, it is necessary to create your identity and obtain an API-KEY from the free cloud service, CAS Cloud, provided by Codenotary. Additionally, you will need to download the CAS executable, which is available for both Unix and Windows operating systems. Once the executable is downloaded, you can proceed to log in to the attestation service using your API key.

Notarization

To notarize a digital asset using Codenotary CAS, follow these steps:

1. Launch the Codenotary CAS executable on your system.
2. Initiate the notarization process by executing the following command: `cas notarize <file>`.
3. Codenotary CAS will calculate a unique cryptographic hash (e.g., using SHA-256) of the asset and record it on the connected blockchain network. This cryptographic hash serves as a digital fingerprint that uniquely identifies the asset.
4. The blockchain network will reach a consensus on the validity of the notarization, ensuring its immutability and tamper-evident nature.
5. Once the notarization process is completed, Codenotary CAS will provide a receipt or proof of notarization. This receipt includes the cryptographic hash of the asset and other relevant metadata. It serves as evidence of the asset's integrity and can be used for future verification purposes.

Verification

To verify a notarized digital asset using Codenotary CAS, follow these steps:

1. Launch the Codenotary CAS executable on your system.
2. Provide the receipt or proof of notarization generated during the notarization process.
3. Initiate the verification process by submitting the receipt for verification. You can use the command: `cas inspect <file>`.
4. Codenotary CAS will retrieve the original cryptographic hash of the asset from the connected blockchain network using the provided receipt.
5. The retrieved hash is then compared to a newly calculated hash of the asset (either the file or the data).
6. If the two hashes match, the asset is considered verified, ensuring that it has not been tampered with since the notarization. Conversely, if the hashes do not match, it indicates that the asset has been altered or tampered with.

By following these steps, you can utilize Codenotary CAS to notarize and verify digital assets, guaranteeing their integrity and immutability through the utilization of blockchain-based technology.

Chapter 3

Integrating MUD in Home Assistant

In this chapter we explore the integration of MUD into Home Assistant which serves as the starting point for our discussion and the focus lies in presenting an extended version of the MUD architecture as stated in [21].

Developers are empowered to specify the endpoints required for their integrations which in turn contribute to the generation of a consolidated gateway-level MUD file. This unique approach enables non-MUD-enabled devices, but in general all kind of plug-ins even the ones that offer only software functionalities, to benefit from MUD compliance when integrated through compatible integrations, all without necessitating modifications to the devices themselves.

3.1 MUD Generator Integration

Despite the absence of native MUD support in Home Assistant, an integration called MUD Generator has been developed to address this limitation. The MUD Generator integration simplifies the process of generating a MUD file by utilizing a provided template. The MUD Generator scans various integrations to identify specific MUD "snippets" associated with each integration. To define their requirements, each integration provides a dedicated JSON file. The MUD integration collects these requirements and combines them into a local MUD file, utilizing a predefined template. This generated MUD file is then signed using the smart home gateway's private key. Once signed, the MUD Generator stores the file in a designated folder accessible through the Home Assistant web server and it also notifies the MUD manager. The MUD Generator Integration plays a vital role in merging MUD snippets contributed by integration developers into a consolidated gateway-level MUD file. This section provides a concise and precise overview of the MUD

Generator Integration, highlighting its key functionality and significance within the Home Assistant ecosystem. The MUD Generator Integration acts as a pivotal link between integration developers and the Home Assistant platform. Its primary objective is to facilitate the submission of MUD snippets, written in a MUD-like format that accurately describe the behavior and requirements of various devices and protocols within a home network. These snippets are then combined to generate a comprehensive MUD file.

3.1.1 MUD Snippets

To ensure consistency and interoperability, integration developers must adhere to a MUD-compliant format when providing MUD snippets. These snippets encapsulate specific rules and characteristics associated with individual devices or protocols, thus enhancing the device security and network access control capabilities of Home Assistant.

We will now discuss the implementation details of the MUD generation functionality in the MUD Generator integration. The code provided is responsible for setting up the MUD Generator platform and creating a button entity that can recreate and expose the MUD file each time it is pressed. This represents a testing version of the MUD Generator integration, designed to facilitate debugging during the development phase. It is important to note that in the definitive version, the MUD Generator will generate the MUD file automatically each time a new integration is added.

The MUD generation implementation starts with the validation of the user's configuration. The `PLATFORM_SCHEMA` defines the expected configuration schema, including optional parameters such as name, network interface and deployment. This schema ensures that the provided configuration is valid and meets the requirements.

The `setup_platform` function is called when setting up the platform. It takes the Home Assistant instance, the configuration, the `add_entities` callback function and the discovery information as input. Inside this function, the configuration parameters are extracted and assigned to the `params` dictionary. If any of the optional parameters are missing from the configuration, default values are set.

The `MUDGeneratorButton` class is defined as a button entity that represents the MUD generator. It takes the Home Assistant instance and the `params` dictionary as input. The `__init__` method initializes the MUD generator button by setting various attributes such as the name, deployment, unique ID and network interface. It also creates an instance of the `MUDGenerator` class.

The `generate_mud_file` method in the `MUDGenerator` class is responsible for generating the MUD file based on a template. It takes the integration list and sign parameters as input. Inside this method, the MUD draft is loaded from a file and MUD rules are added. The MUD file is then checked for changes and if any changes are detected, the *last-update* timestamp is updated and the MUD file is written.

The `_add_mud_rules` method in the `MUDGenerator` class adds ACLs (Access Control Lists) to the MUD object. It takes the integration list as input and retrieves MUD snippets from the manifest of each integration. It also traverses the *custom components* and *default components* — which are the ones officially present in the Home Assistant release — directories to find MUD snippets and adds them to the MUD object. Searching among default components was implemented considering a future full compatibility of Home Assistant with MUD. In this way, even MUD snippets of these default components will be included in the generated MUD file.

The `_add_rules_to_draft` method in the `MUDGenerator` class adds the extracted MUD rules from a snippet file to the MUD draft. It parses the snippet file, extracts the necessary rules, and adds them to the appropriate sections of the MUD draft, such as the *from-device* policy, *to-device* policy, and *ACLs*.

Overall, the MUD Generator implementation allows the user to configure the MUD generator integration and generate the MUD file based on the provided configuration. The integration automatically updates the MUD file periodically and exposes it through the specified interface. The MUD rules are extracted from mud-snippets files present in the integrations manifest, ensuring that the generated MUD file contains the necessary rules to enforce network security policies for IoT devices.

3.2 Extended MUD Architecture

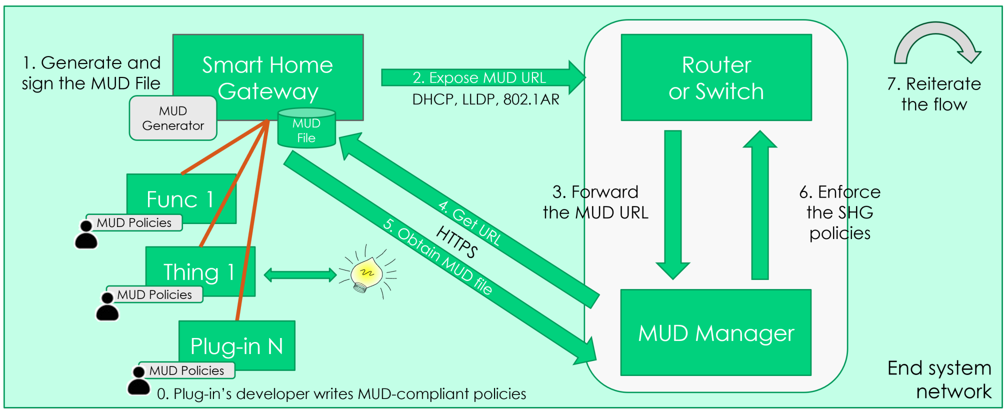


Figure 3.1: Extended MUD Architecture

In this section we aim to describe the extended MUD architecture, which will enable plug-in developers to define a set of MUD-compliant requirements. Its components are described in detail in Chapter 2.2.

The primary objective of this extended architecture is to allow plug-in developers to specify a comprehensive set of MUD requirements enanching the original idea of MUD files which are provided by the device manufacturers itself. These requirements will then be utilized by the Smart Home Gateway to automatically generate a gateway-level MUD file.

This architecture leverages various components. It employs an OpenWrt [22] router running osMUD [23], an open-source MUD manager. The osMUD component only supports DHCP [11] as the notification approach specified in the MUD RFC. The Home Assistant (Hass) component plays a crucial role in sending DHCP requests to the router, including the appropriate MUD URL. This URL points to the Hass instance, which hosts a web server capable of exposing the generated MUD file, along with the Hass dashboard.

Once osMUD retrieves the MUD file, it verifies the associated signature and enforces the specified policies by configuring the OpenWrt firewall using *iptables*. This ensures that the communication rules and security policies remain up-to-date.

To keep the MUD file current with all available MUD-enabled integrations, the process is repeated whenever users add new integrations to Home Assistant. This involves restarting the SHG to activate the new integrations, prompting the retrieval of an updated MUD file. When osMUD obtains the new file, it removes the old firewall rules and applies the new policies accordingly.

In the context of this gateway-based MUD architecture, collaboration among developers is crucial. Developers are responsible for specifying the endpoints required for their integrations and adhering to the MUD standard by creating MUD-compliant files for each integration. For developers who may not possess an in-depth understanding of these standards, tools like MUD Maker[24] can be utilized to simplify the creation of MUD snippets.

It is important to note that this architecture assumes that every smart home gateway plug-in can benefit from MUD, even if the plugin integrates only new software functionalities. This inclusive approach aims to enhance the overall security of the SHG and, consequently, the entire smart home system. Additionally, by enabling plug-in developers to act on behalf of device manufacturers, the number of MUD-compliant devices supported within a smart home environment can be increased.

By embracing this collaborative architecture, users of Home Assistant can enhance their security and enforce communication restrictions based on MUD files provided by integration developers. The close cooperation between developers and the MUD manager ensures a more secure and controlled smart home environment.

Chapter 4

Authenticating Integrations and MUD Snippets

In this chapter we will discuss the authentication mechanism implemented in the MUD Generator integration, which significantly enhances the reliability and trustworthiness of the architecture described in Chapter 3. The authentication mechanism provides an additional layer of assurance by verifying the integrity and origin of integrations (and their corresponding MUD snippets) present in Home Assistant. This ensures that only trusted and authenticated integrations are utilized within the system thereby increasing the overall security of the smart home environment.

4.1 Introduction

The Home Assistant platform's flexibility and openness introduce potential security vulnerabilities, particularly in relation to third-party plug-ins. These plug-ins, available through platforms like Home Assistant Community Store (HACS) or directly from the Internet, can pose risks such as the inclusion of malicious code or vulnerabilities that compromise the overall security of the smart home system. To address these security concerns, the goal of this thesis is to introduce in this architecture an authentication and verification mechanism specifically tailored for the MUD snippets and their associated integrations within the Home Assistant system. This process plays a crucial role in maintaining the integrity and security of the generated MUD file.

This chapter explores the integration notarization process using the CodeNotary CAS service and the subsequent authenticity check performed by the MUD Generator integration. We will also discuss the severity levels associated with the authenticity check and their corresponding actions. Through these mechanisms, Home Assistant ensures the trustworthiness and integrity of integrations, thereby enhancing the reliability and security of the smart home system.

4.2 Integration Notarization with CAS

As explained in Section 2.3.3, integration developers are required to notarize their integrations using the CodeNotary CAS service. This notarization process ensures that the MUD snippet and its associated integration remain unchanged and authentic over time. By notarizing the integration, a unique cryptographic hash is generated and associated with it.

The integration notarization process involves the following steps:

1. The integration developer, after obtaining its private API-KEY from CAS, must login into the CAS service.
2. The integration developer submits the MUD snippet and its associated integration to the CodeNotary CAS service.
3. The CAS service calculates the cryptographic hash value of the submitted code and configuration.
4. The CAS service associates the generated hash value with the integration, establishing a verifiable link between the code and its hash.

By performing this notarization step, the authenticity and integrity of the integration are ensured. The unique hash value serves as a digital fingerprint that can be used for subsequent verification.

4.3 Integration Authenticity Check

The MUD Generator integration provides users with the option to perform authenticity checks on the integrations present in Home Assistant. These checks verify the authenticity and integrity of the integrations, ensuring that they have not been tampered with or modified maliciously. The severity level for the authenticity check is selected by the user, giving them control over how the security mechanism of authentication is handled. This allows users to customize the level of strictness according to their preferences and requirements, providing flexibility in aligning the security posture of their Home Assistant system.

This authentication process serves multiple purposes:

- **MUD Snippet Integrity Check:** Similarly, the CAS service verifies the integrity of the MUD snippets, guaranteeing their authenticity and preventing the inclusion of tampered or malicious snippets.

- **Integrity Check:** The CAS service validates the integrity and reliability of the submitted integrations, ensuring that they have not been tampered with or modified maliciously.
- **Binding Verification:** The authentication check also includes verifying that each integration is correctly associated with its appropriate MUD snippet. This verification step safeguards against the inclusion of misconfigured or unauthorized snippets.

4.4 Severity Levels

The authenticity check in the MUD Generator integration operates based on severity levels. These levels determine the strictness of the check and the corresponding actions taken when an integration fails the check. The severity levels and their associated actions are as follows:

- **High Severity:** Integrations failing the authenticity check with high severity are disabled within Home Assistant. Additionally, their MUD snippets are not considered valid and the associated communication rules are disregarded. This level ensures a strict security posture by completely disabling integrations that fail the authenticity check. Users are immediately notified about the failed check and the resulting action indicating that the integration is no longer trusted and should be investigated further.
- **Medium Severity:** Integrations failing the authenticity check with medium severity are not disabled. However, the MUD snippet associated with these integrations is not considered valid, and the communication rules specified within it are ignored. This level ensures that potentially compromised integrations do not open any network communication based on a potentially untrustworthy MUD snippet. Users are notified about the failed check, emphasizing the importance of investigating and resolving the integrity issue.
- **Low Severity:** Integrations failing the authenticity check with low severity remain enabled within Home Assistant. The associated MUD snippet is still considered valid and the communication rules specified within it are enforced. This level provides a more lenient approach, allowing integrations to continue functioning even if their authenticity is in question. However, users are notified about the failed check, providing transparency and awareness regarding the integrity of the integration.

The selected severity level applies to all integrations within Home Assistant and determines the level of strictness in the authenticity check. It is important to note that these severity levels are configurable by the user, allowing for flexibility in aligning the security posture with individual preferences and requirements.

Table 4.1 summarizes the expected behaviour of the MUD Integration according to the chosen severity level.

Table 4.1: Severity levels and related behaviour

Severity Level	**Plugin Status**	**MUD Snippet**
Low	Enabled	Enabled
Medium	Enabled	Disabled
High	Disabled	Disabled

In the next section, we will delve into the implementation details of the authentication mechanism employed by the MUD Generator integration, exploring its working mechanisms and how it integrates with other components within the Home Assistant ecosystem.

4.5 Expanding the MUD Integration

In this section, we will delve into the implementation details of the MUD Generator Integration, providing a comprehensive explanation of the underlying code and its functionality.

The MUD Generator Integration follows a modular and extensible code architecture, making use of the Home Assistant framework's capabilities. The integration code is organized into separate modules and files, each serving a specific purpose and contributing to the overall functionality.

4.5.1 Functionalities Overview

The authentication implementation diagram in Figure 4.1 provides an overview of the authentication flow and the relevant components involved.

Setup Function

The `async_setup` function is responsible for setting up the integration within the Home Assistant framework. It takes two parameters: `hass`, which represents the Home Assistant instance, and `config`, which holds the configuration data.

Inside the `async_setup` function, the severity level is determined based on the configuration provided in the `config` parameter. As previously explained, severity level determines how the integration will respond to certain events and conditions.

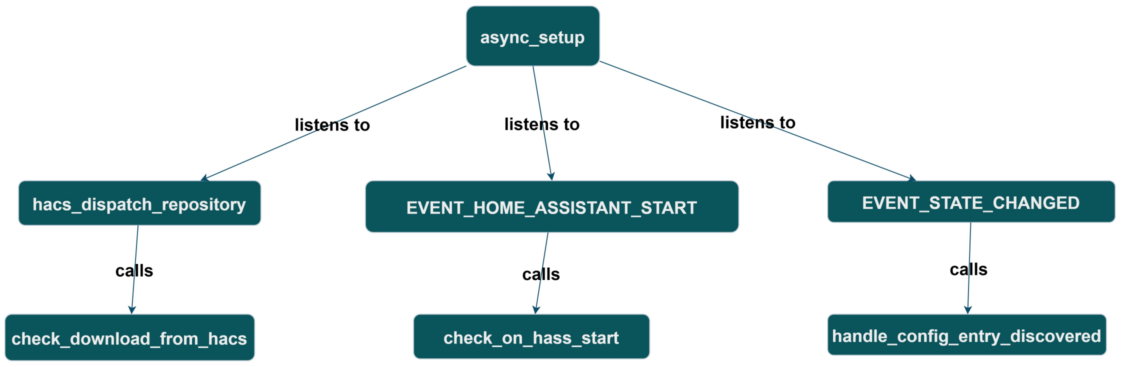


Figure 4.1: Authentication implementation diagram

Download Time

One aspect of the integration is monitoring the download time when a user downloads an integration from the Home Assistant Community Store (HACS). To handle this, a partial callback function, `check_download_from_hacs`, is defined. It takes the Home Assistant instance, `hass`, as a parameter.

The partial callback function is connected to the `hacs_dispatch_repository` event using `async_dispatcher_connect`. This allows the function to be called whenever a download event occurs in HACS.

Within the `check_download_from_hacs` function, the data associated with the download event is extracted and processed. If the the action is an installation, the authenticity check is performed.

Startup Time

Another aspect of the integration involves monitoring the startup time of Home Assistant. This is done by listening to the `EVENT_HOMEASSISTANT_START` event. When this event occurs, the `check_on_hass_start` function is called.

The `check_on_hass_start` function takes the Home Assistant instance, `hass`, and the severity level as parameters. It retrieves the list of custom components and excludes the MUD Generator integration and the HACS integration.

Each integration in the list is then checked using the `check_integration` function. If an integration fails the check, the severity level determines the appropriate action to be taken. For high severity, the integration is disabled, while for low or medium severity, a notification is sent to the user.

Runtime Integration Addition

Furthermore, the MUD Generator provides support for detecting and handling runtime additions of integrations via the web GUI. This is accomplished by listening to the `EVENT_STATE_CHANGED` event and calling the `handle_config_entry_discovered` function.

The `handle_config_entry_discovered` function takes the Home Assistant instance, `hass`, the severity level, and the event data as parameters. It extracts the domain and integration information from the event data and performs checks using the `check_integration` function.

Based on the severity level, the integration is either disabled or a notification is sent to the user if it fails the check.

Periodic check at runtime

The periodic check at runtime is another important aspect of the MUD Generator integration. It involves performing periodic checks on the integrations running in the system to ensure they have not been tampered at runtime. This periodic check is implemented through a scheduled task that runs at regular intervals.

The scheduled task, named `perform_periodic_check`, is registered using Home Assistant's scheduler. It is configured to execute every X minutes, where X is a configurable parameter set by the user.

Within the `perform_periodic_check` function, the integration retrieves the list of active integrations in Home Assistant. For each integration in the list, the function calls the `check_integration` function to evaluate its current state and compliance with the predefined criteria.

If an integration fails the check, the severity level determines the appropriate action to be taken. For high severity, the integration is disabled to prevent further issues or potential system instability. For low or medium severity, a notification is sent to the user, providing information about the specific integration and the nature of the issue encountered.

After evaluating all active integrations, the `perform_periodic_check` function completes its execution. The scheduler ensures that the function will be called again at the next scheduled interval, allowing for continuous monitoring and maintenance of the integrations during runtime.

By implementing the periodic check at runtime, the integration provides a proactive approach to ensure the stability and reliability of the integrations running within the Home Assistant system, helping users identify and resolve potential issues in a timely manner.

4.5.2 Integrity Check Implementation

The integrity check plays a crucial role in ensuring the trustworthiness and reliability of software systems. In the context of home assistant integrations, integrity checks are used to verify the authenticity and integrity of the integration's files. By performing integrity checks, Home Assistant can ensure that the integration's files have not been tampered with and that they are genuine.

The three functions we will discuss are part of an integrity checking mechanism for home assistant integrations. These functions provide an alternative method for performing integrity checks and rely on the SHA-256 [20] hashing algorithm and the CAS Codenotary tool.

Function: `check_integration`

The `check_integration` function serves as the entry method for performing an integrity check on a Home Assistant integration.
It takes two parameters: `integration_name`, which is the name of the integration and `files_to_be_verified`, which is a list of file names to be verified (default is `None`). The `files_to_be_verified` parameter allows flexibility for integration developers who may want to specify a subset of files within their integration that need to be checked for integrity rather than performing the check on all files by default.

The function first calls the `hash_folder` function to calculate the SHA-256 hash value of the integration. This hash value is then passed to the `verify_hash` function to check if it was notarized.

Listing 4.1: Function: `check_integration`

```
def check_integration(integration_name, files_to_be_verified):
    """
    Function to perform integrity check.
    """
    hash_value = hash_folder(integration_name, files_to_be_verified)
    return verify_hash(hash_value)
```

Function: `hash_folder`

The `hash_folder` function calculates the SHA-256 hash value of a folder or a specific set of files within the integration.
It takes two parameters which are the same as the function above.

The function starts by constructing the path to the integration using the `integration_name` parameter. It then initializes a SHA-256 hash object.

If no specific files are specified (i.e., `files_to_be_verified` is `None` or an empty list), the function recursively walks through all the files in the integration folder.

For each file, it opens the file in binary mode, reads its contents, and updates the hash object with the file contents.

If specific files are provided, the function searches for each file recursively within the integration folder. Once a file is found, it reads its contents and updates the hash object. If a file is not found, a `FileNotFoundError` is raised.

Finally, the function returns the hexadecimal representation of the hash value.

Listing 4.2: Function: `hash_folder`

```
def hash_folder(integration_name, files_to_be_verified=None):
     """
     argument:
         integration_name: Name of the integration
         files_to_be_verified: List of file names to be verified,
    default is None
     This function builds a SHA-256 hash object which accumulates
     the hash values of all the file contents in the list of files
    provided,
     resulting in a hash value that is a function of the binary
    contents of those files.
     """

     integration_path = os.path.join(REAL_CUSTOM_COMPONENT_DIR,
    integration_name)

     sha256 = hashlib.sha256()

     # If no files are specified, hash all files in the folder
     if files_to_be_verified is None or files_to_be_verified == []:
         # Walk through all the files and subfolders in the given
    folder
         for root, _, files in os.walk(integration_path):
             for file in files:
                 # Get the full path of the file
                 file_path = os.path.join(root, file)

                 # Read the file contents and update the hash
                 with open(file_path, "rb") as file:
                     sha256.update(file.read())
     else:
         # Hash only the specified files, searching recursively
    through subfolders
         for file in files_to_be_verified:
             # Find the file by recursively searching through all
    subdirectories
             for root, _, files in os.walk(integration_path):
                 if file in files:
                     # Get the full path of the file
                     file_path = os.path.join(root, file)
```

```

                    # Read the file contents and update the hash
                    with open(file_path, "rb") as file:
                        sha256.update(file.read())
                    break
            else:
                # If the file is not found, raise an error
                raise FileNotFoundError(
                    f"File '{file}' not found in folder '{
    integration_name}'"
                )

    # Return the hash value
    return sha256.hexdigest()
```

Function: `verify_hash`

The `verify_hash` function executes the CAS Codenotary executable to inspect the given hash value and determine if it was notarized.
It takes a single parameter, `hash_value`, which is the hash value of the integration. The function first logs in to the CAS Codenotary. If the login is successful, it logs the message "Logged In!". Otherwise, it logs an error message. Next, it executes the CAS Codenotary `inspect` command with the provided API key and the given `hash_value`. The function captures the output and checks if there was any error. If there was no error, it processes the output to determine the verification status and the signer ID. If the verification status is "TRUSTED", the function logs the signer ID and the status as verified. Otherwise, it logs the signer ID and the status as not verified. If there was an error in the execution of the CAS command or the hash value was not notarized, the function logs an appropriate error message. Finally, the function returns `True` if the integration is verified and `False` otherwise.

Listing 4.3: Function: `verify_hash`

```
def verify_hash(hash_value):
    """
    argument: hash of the integration ( the hash value, which is
    notarized, could be inserted in the manifest.json )
    This function just executes the CAS Codenotary executable to
    inspect the hash and see if it was notarized.
    """

    process = subprocess.run(
        [
            "/config/custom_components/mud-generator/cas",
            "login",
            "--api-key",
            f"{CAS_API_KEY}"
        ],
        capture_output=True,
        check=False,
    )

    stderr = process.stderr.decode()
    if stderr == "":
        logging.info("Logged In!")
    else:
        logging.info("ERRORE = %s", stderr)


    process = subprocess.run(
        [
            "/config/custom_components/mud-generator/cas",
            "--api-key",
            f"{CAS_API_KEY}",
            "inspect",
            "--hash",
            f"{hash_value}",
        ],
        capture_output=True,
        check=False,
    )

    stderr = process.stderr.decode()
    logging.info(stderr)

    if stderr == "":
        stdout = process.stdout.decode()
        if "0 notarizations found" in stdout:
            logging.info(
                "%s \033[31m : not verified! ( 0 notarizations found ) \033[0m\n",
                hash_value,
            )
            return False
        else:
            matched_lines = [line for line in stdout.split("\n") if "Status" in line]
            status = matched_lines[0].replace("\t", "").split(":")[1]
            matched_lines = [line for line in stdout.split("\n") if "SignerID" in line]
            signerid_encoded = matched_lines[0].replace("\t", "").split(":")[1]
            decoded_bytes = base64.b64decode(signerid_encoded)
            signerid_decoded = decoded_bytes.decode("utf-8")
            logging.info("SignerID: \033[34m %s \033[0m", signerid_decoded)
            logging.info("Status: \033[32m %s \033[0m", status)
            if status == "TRUSTED":
                logging.info("%s \033[32m : verified! \033[0m\n", hash_value)
                return True
            else:
                logging.info("%s \033[31m : not verified! \033[0m\n", hash_value)
                return False
    else:
        if "not notarized" in stderr:
            logging.info("\033[31m : not verified! \033[0m")
            return False
        else:
            logging.info("Error in the execution of CAS command!")
            return False
```

By using the `check_integration` function, you can perform an integrity check on your Home Assistant integration and verify if it was notarized using the CAS Codenotary tool.

In conclusion, the integration authentication mechanism described in this section provides essential functionalities to maintain the integrity and reliability of integrated components within the Home Assistant framework. By monitoring download time, startup time, runtime integration addition and conducting periodic checks at runtime, the mechanism ensures ongoing verification and proactive management of the integrations.

If you are interested in exploring the detailed code implementation, you can find it on GitLab[25]. Please note that the repository is currently private.

4.6 Use Case Examples for the MUD Generator Integration

Here are some possible use case examples for the MUD Generator integration considering the different integrity check points and severity levels. These examples highlight the behavior of the integration and its impact on the system's security.

4.6.1 Integrity Check at Download Time:

Use Case 1:

1. The user downloads an integration from HACS.

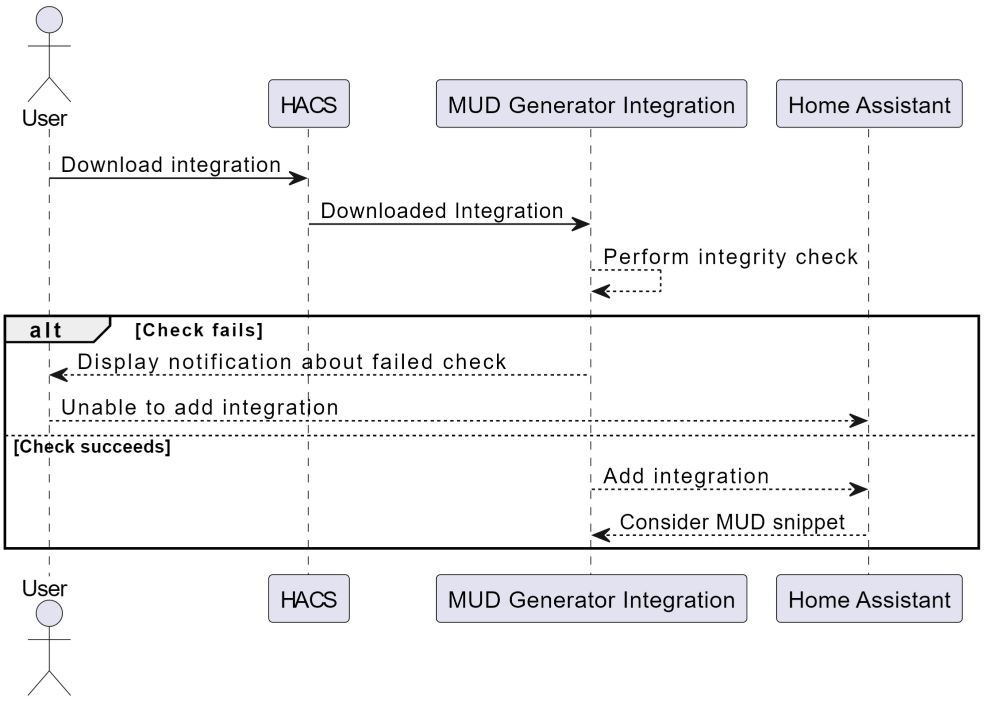


Figure 4.2: Use Case 1 diagram

2. The MUD Generator integration performs an integrity check on the downloaded integration.
3. If the check fails:
   (a) The user is unable to add the integration to Home Assistant.
   (b) A notification is displayed to inform the user about the failed check.
4. If the check succeeds:
   (a) The integration is added to the system.
   (b) The MUD snippet associated with the integration is considered.

4.6.2 Integrity Check at Startup Time:

Use Case 2:

1. The user starts Home Assistant.
2. The MUD Generator integration performs integrity checks on all the integrations in the `custom_components` directory.

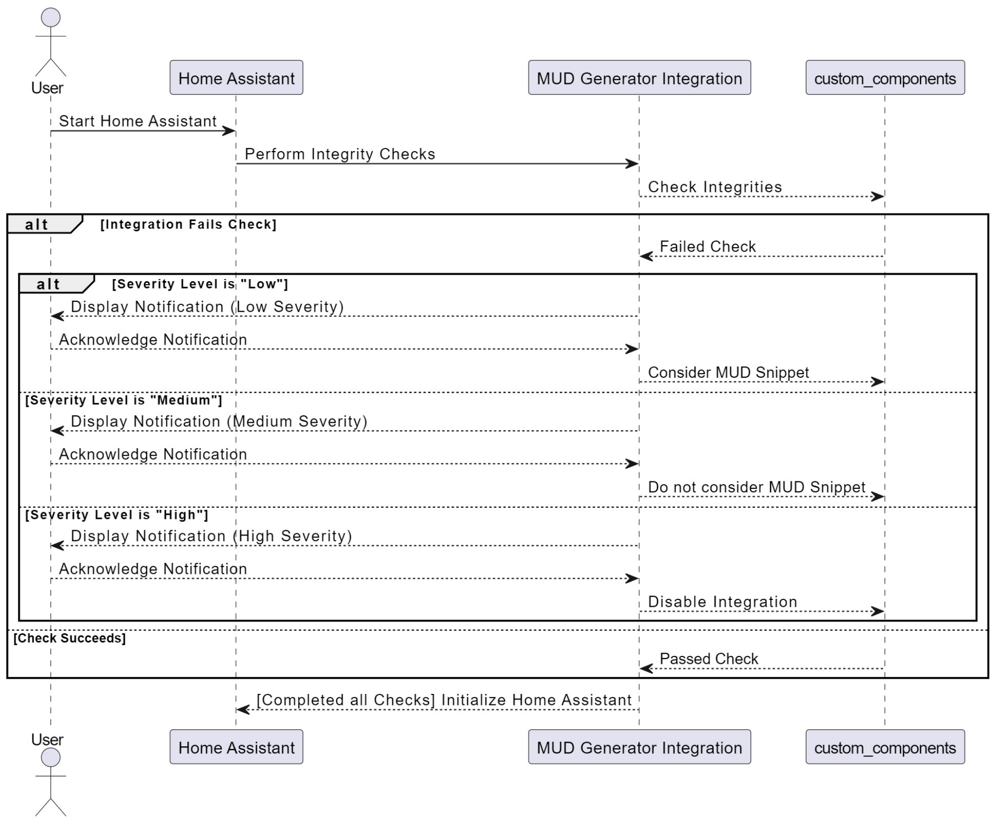

User
Home Assistant
MUD Generator Integration
custom_components
Start Home Assistant
Perform Integrity Checks
Check Integrities
alt [Integration Fails Check]
Failed Check
alt [Severity Level is "Low"]
Display Notification (Low Severity)
Acknowledge Notification
Consider MUD Snippet
[Severity Level is "Medium"]
Display Notification (Medium Severity)
Acknowledge Notification
Do not consider MUD Snippet
[Severity Level is "High"]
Display Notification (High Severity)
Acknowledge Notification
Disable Integration
[Check Succeeds]
Passed Check
[Completed all Checks] Initialize Home Assistant

Figure 4.3: Use Case 2 diagram

3. If an integration fails the check:
   (a) If the severity level is set to "Low":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is not disabled.
      iii. The MUD snippet associated with the integration is considered.
   (b) If the severity level is set to "Medium":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is not disabled.
      iii. The MUD snippet associated with the integration is not considered.
   (c) If the severity level is set to "High":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is disabled.
      iii. The MUD snippet associated with the integration is not considered.
4. Home Assistant is fully initialized after all integrity checks are completed.

4.6.3 Integrity Check at Runtime:

Use Case 3:

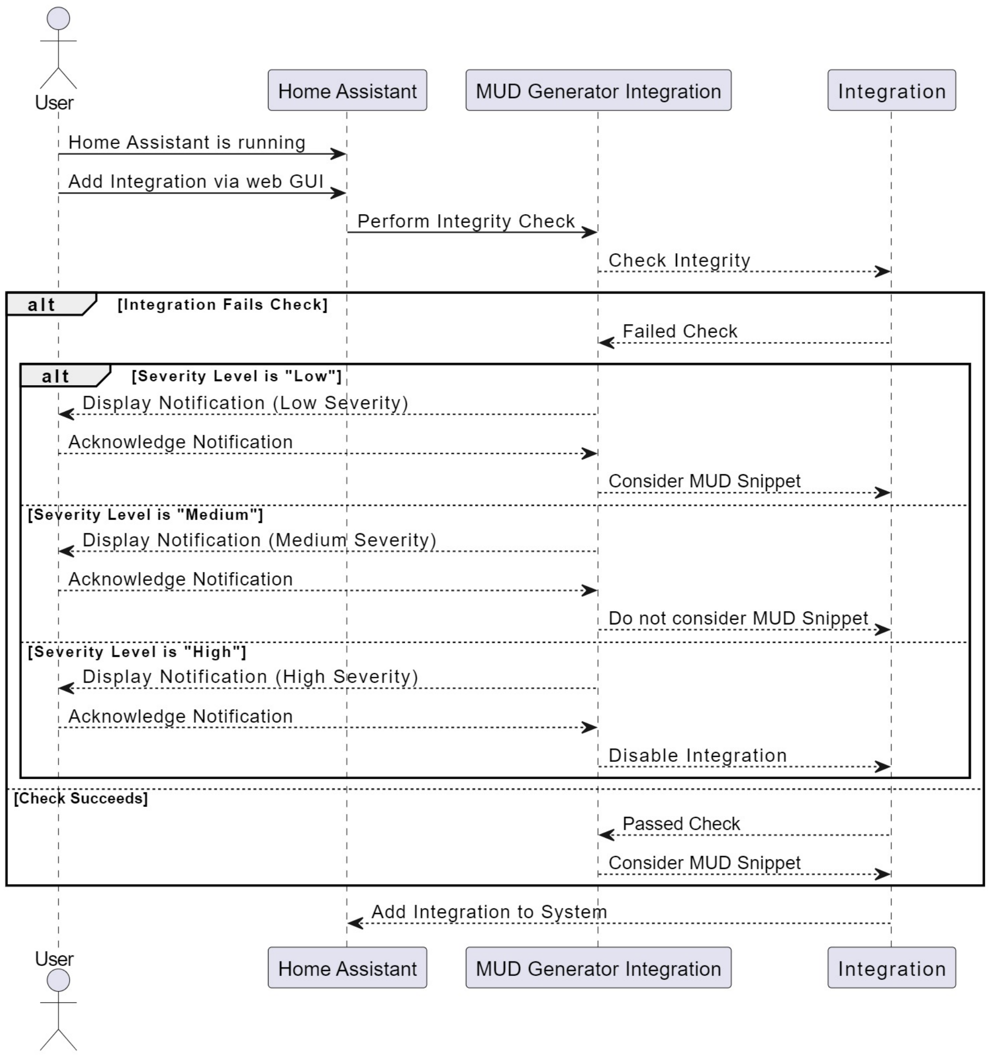

User
Home Assistant
MUD Generator Integration
Integration
Home Assistant is running
Add Integration via web GUI
Perform Integrity Check
Check Integrity
alt [Integration Fails Check]
Failed Check
alt [Severity Level is "Low"]
Display Notification (Low Severity)
Acknowledge Notification
Consider MUD Snippet
[Severity Level is "Medium"]
Display Notification (Medium Severity)
Acknowledge Notification
Do not consider MUD Snippet
[Severity Level is "High"]
Display Notification (High Severity)
Acknowledge Notification
Disable Integration
[Check Succeeds]
Passed Check
Consider MUD Snippet
Add Integration to System

Figure 4.4: Use Case 3 diagram

1. Home Assistant is running.
2. The user adds an integration via the web GUI.
3. The MUD Generator integration performs an integrity check on the integration to be added.

4. If the check fails:
   (a) If the severity level is set to "Low":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is not disabled.
      iii. The MUD snippet associated with the integration is considered.

   (b) If the severity level is set to "Medium":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is not disabled.
      iii. The MUD snippet associated with the integration is not considered.

   (c) If the severity level is set to "High":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is disabled.
      iii. The MUD snippet associated with the integration is not considered.

5. If the check succeeds:
   (a) The integration is added to the system.
   (b) The MUD snippet associated with the integration is considered.

Use Case 4:

1. Home Assistant is running.
2. The MUD Generator integration periodically performs integrity checks on all active integrations.
3. If an integration fails the check:
   (a) If the severity level is set to "Low":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is not disabled.
      iii. The MUD snippet associated with the integration is considered.

   (b) If the severity level is set to "Medium":
      i. A notification is displayed to inform the user about the failed check.
      ii. The integration is not disabled.
      iii. The MUD snippet associated with the integration is not considered.

   (c) If the severity level is set to "High":

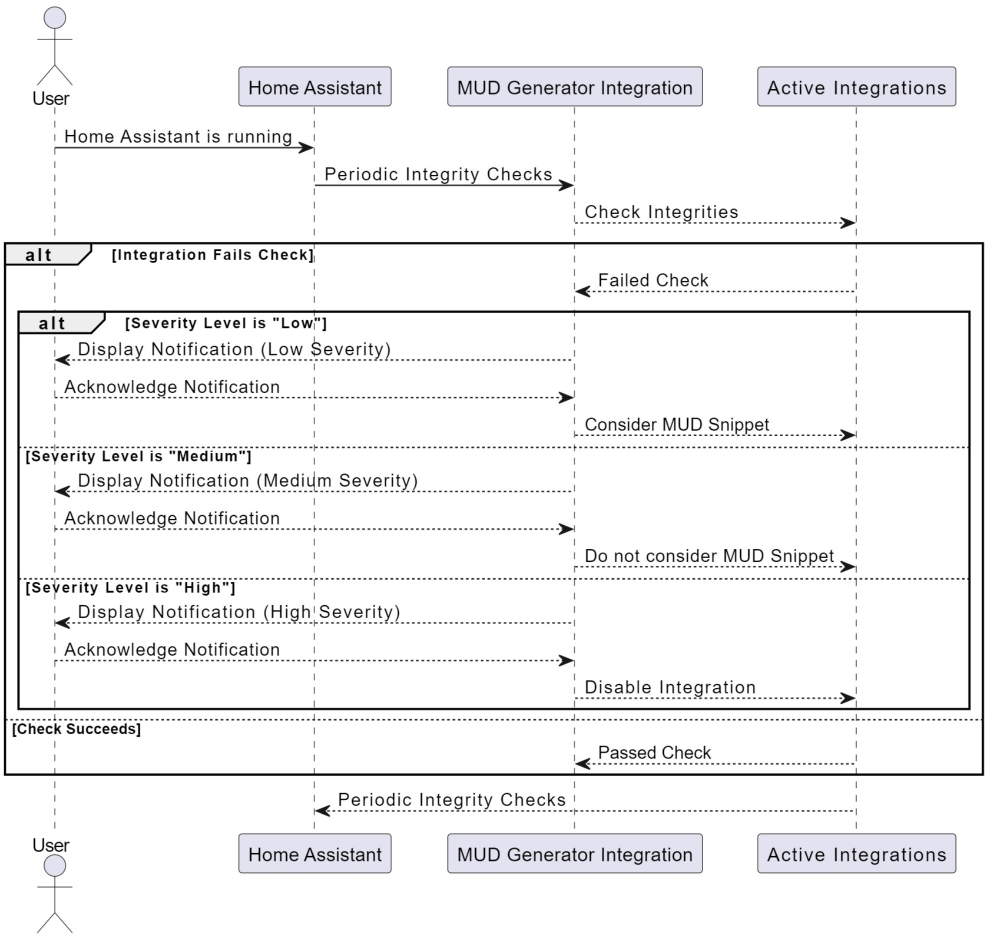

User
Home Assistant
MUD Generator Integration
Active Integrations
Home Assistant is running
Periodic Integrity Checks
Check Integrities
alt [Integration Fails Check]
Failed Check
alt [Severity Level is "Low"]
Display Notification (Low Severity)
Acknowledge Notification
Consider MUD Snippet
[Severity Level is "Medium"]
Display Notification (Medium Severity)
Acknowledge Notification
Do not consider MUD Snippet
[Severity Level is "High"]
Display Notification (High Severity)
Acknowledge Notification
Disable Integration
[Check Succeeds]
Passed Check
Periodic Integrity Checks

Figure 4.5: Use Case 4 diagram

i. A notification is displayed to inform the user about the failed check.
ii. The integration is disabled.
iii. The MUD snippet associated with the integration is not considered.

4. The periodic integrity checks ensure that the system remains secure even during runtime.

These use cases showcase the behavior of the MUD Generator integration in different scenarios, considering the integrity checks at download time, startup time, and runtime. The severity level determines the actions taken when an integration fails the check, such as disabling the integration and excluding the associated MUD snippet.

4.7 Experimental Results

4.7.1 Experimental Setup

The experimental setup involved evaluating the performance impact of incorporating an authentication mechanism into Home Assistant for authentication and verification of integrations. The experiment was conducted using specific hardware and software configurations. Specifically, Home Assistant OS (version 2023.7.1) was installed on a Raspberry Pi 3B, configured with a set of 23 integrations (70% of them were notarized by hypothetical developers), the previously presented MUD Generator integration, HACS version 1.32.1 and Codenotary CAS version 1.0.3. Alongside the first Raspberry, there was another one acting as a router equipped with OpenWrt 17.01.6. On the same device, was also installed the Open Source MUD Manager (osMUD).

The experiment aimed to measure the startup time of Home Assistant with and without the integration of the authentication mechanism. We recorded the time it took for Home Assistant to start up under both conditions. The measurements were taken using a stopwatch and each condition was repeated multiple times for accuracy. At the end we computed the average.

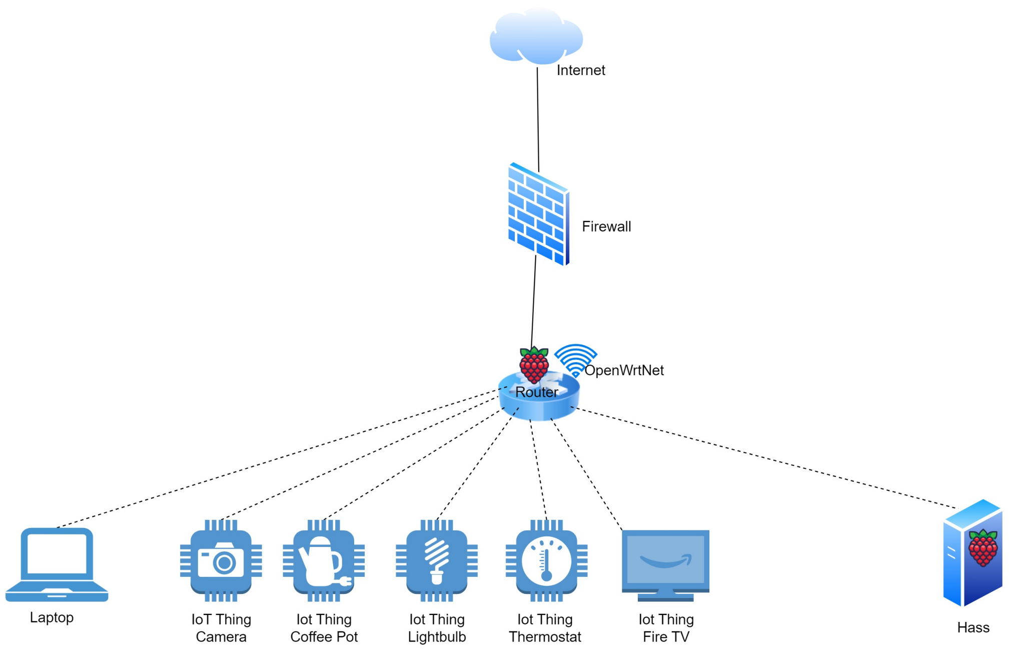


Figure 4.6: Conceptual Network Architecture

Router (OpenWrt + osMUD + dhcpmasq)

One of the Raspberry Pi devices is configured as a router, utilizing the OpenWrt operating system. It is connected to the Polytechnic University of Turin Ethernet network via an Ethernet cable and it expose a public IP address. This setup establishes internet connectivity for the network. The router also provides a Wi-Fi network named "OpenWrtNet". By connecting to this network, users gain access to the router's dashboard and the Home Assistant dashboard.

The router is equipped with osMUD, a MUD Manager and a modified version of dnsmasq, which serves as a DHCP server required by osMUD to analyze DHCP requests.

Smart Home Gateway

The second Raspberry Pi hosts the latest version of Home Assistant OS.

Within Home Assistant OS, along with a few test integrations and the integration of new physical devices it consists also of the MUD Generator integration that was implemented to enable MUD-based device management and integrations/mud authentication.

It is important to note that while the network architecture presented various physical devices, the majority of these devices have not been physically integrated with Home Assistant. However, this is not so important because the focus of this thesis lies in the notarization of integration software rather than of the devices themselves.

4.7.2 Results

The experimental results revealed that the integration of the authentication mechanism increased the startup time of Home Assistant. Without it, Home Assistant took an average of 5 minutes and 1 seconds to start up. However, with the integration of the authentication mechanism, the startup time increased to an average of 7 minutes and 22 seconds.

Table 4.2: Home Assistant Startup Times

Authentication	Fastest Run				Slowest Run	Average (minutes)
Disabled	04:26	04:28	04:45	05:29	06:01	05:01
Enabled	06:49	07:00	07:26	07:43	07:54	07:22

The results showed that the integration of the authentication mechanism increased the startup time of Home Assistant by approximately 46.52% (from an average of 5 minutes and 1 second to 7 minutes and 22 seconds). A comparison of the startup times highlighted the impact on performance, with the authentication mechanism adding an additional 2 minutes and 21 seconds to the startup time. The observed increase in startup time was attributed to factors such as network latency, computational overhead for authentication and verification processes and the time taken to perform checks on available integrations. However, during the normal execution of Home Assistant, the MUD Generator integration does not noticeably impact performance.

It is important to note the limitations of the experiment, including the specific hardware configuration used and the dataset of integrations, which may not represent all possible scenarios. It should be acknowledged that utilizing hardware such as the Raspberry Pi 4, as suggested by Home Assistant, could potentially yield better performance results. Additionally, it is crucial to consider the dataset of integrations, as the number of integrations present within the platform directly impacts the number of checks that need to be performed, potentially affecting performance.

Fortunately, in the case of a smart home gateway, it is not frequently necessary to reboot the system. Therefore, the longer startup time observed during the experiment does not significantly impact the overall performance of the gateway during regular operation. Once the system is up and running, the increased startup time becomes less relevant in the context of continuous usage or in a stable operational state.

Overall, the experiment provided insights into the performance impact of incorporating an authentication mechanism into Home Assistant and highlighted the trade-off between enhanced security benefits and increased startup time.

Chapter 5

Conclusions

Based on the foundation of the MUD-based architecture in Home Assistant, this Master's thesis has made significant advancements by introducing an authentication and verification mechanism specifically tailored for MUD snippets and their associated integrations. This work aims to address the potential security vulnerabilities that arise from the platform's flexibility and openness, particularly concerning third-party plug-ins.

The MUD-based architecture in Home Assistant serves as the starting point for our research. It leverages the concept of MUD files which describe the behavior and requirements of devices and protocols within a home network. By integrating MUD into Home Assistant, developers can specify the necessary endpoints for their integrations, leading to the generation of a consolidated gateway-level MUD file. This unique approach allows non-MUD-enabled devices and various plug-ins, including those offering only software functionalities, to benefit from MUD compliance without requiring modifications to the devices themselves.

However, the Home Assistant platform's openness also introduces potential security vulnerabilities, particularly when it comes to third-party plug-ins. These plug-ins, available through platforms like Home Assistant Community Store (HACS) or directly from the Internet, can pose risks such as the inclusion of malicious code or vulnerabilities that compromise the overall security of the smart home system.

To address these security concerns, our thesis introduces an authentication and verification mechanism specifically tailored for the MUD snippets and their associated integrations within the Home Assistant system. This mechanism plays a crucial role in maintaining the integrity and security of the generated MUD file.

The authentication and verification mechanism is incorporated within the Home Assistant framework and continuously monitors the environment at different stages, such as download time and startup time. It ensures continuous verification, proactive management, and timely resolution of potential security issues within the integrations during runtime.

By developing this authentication and verification mechanism for Home Assistant plug-ins, this thesis significantly enhances the security and reliability of smart home systems. Users can confidently incorporate third-party plug-ins into their setups, knowing that only validated and trusted plug-ins are utilized, thereby maintaining the required security standards.

In conclusion, this work builds upon the foundation of the MUD-based architecture in Home Assistant and introduces an authentication and verification mechanism to enhance the overall security and reliability of smart home systems. By addressing the security concerns related to third-party plug-ins, we provide users with a more secure environment for managing their smart homes. The successful implementation of this mechanism opens up avenues for further exploration and innovation, ultimately leading to safer and more trustworthy smart home ecosystems.

5.1 Future works

5.1.1 Identity Registration Service

In order to enhance the overall functionality and security of the system, a potential avenue for future development is the implementation of an Identity Registration System for developers. This system aims to provide developers with the ability to create their own unique identities which can be utilized for various purposes within the ecosystem.

To facilitate this idea, a dedicated "Registration Service" can be established. The primary objective of this service is to allow developers to create their identities while incorporating varying levels of authentication to validate their trustworthiness.

The proposed "Registration Service" will offer developers a comprehensive registration process, allowing them to provide essential information such as their name, surname, email, and phone number. Additionally, the service can offer various authentication mechanisms, including strong identification options such as identity document verification.

During registration, developers can opt for different authentication levels based on the provided information. For example, providing basic details like name, surname, email, and phone number can be considered a standard authentication level. However, if developers choose to include a scan of their identity document, such as an identity card or driver’s license, this can be regarded as a higher authentication level due to the increased assurance of their identity.

Upon successful registration, the CAS API-KEY will be sent to their registered email address. Once developers have obtained their API-KEY, they can proceed to utilize the CAS functionalities as previously described. However, the key distinction now is that our service will have access to the developer’s identity information.

This added knowledge allows for enhanced verification and validation processes within the ecosystem.

For instance, the MUD Manager can cross-reference the SignerID in the received Snippets with our service's database. If a Snippet is notarized but the associated SignerID is not present in our service's records it indicates that the asset was notarized by a third party who has not registered with our service. This additional layer of identity verification can further strengthen the overall security and trust within the system.

By implementing an Identity Registration System for developers, we can enhance the accountability and transparency of the ecosystem. This system provides a clear link between developers, their identities and their actions within the platform. Furthermore, it allows for more comprehensive tracking and management of developer interactions fostering a secure and trustworthy environment for all participants involved.

5.1.2 Vulnerability analysis of integrations

Another area of potential future work involves implementing a vulnerability scanning mechanism for added Home Assistant integrations. This mechanism aims to proactively identify and address security vulnerabilities within integrations, ensuring a safer and more secure environment for users.

The proposed scanning process would involve regularly analyzing the code and configurations of newly added Home Assistant integrations. This analysis would be performed using a combination of static code analysis, dynamic code testing, and vulnerability database lookups.

Static code analysis involves examining the integration's source code without executing it. This analysis can identify potential security flaws, such as insecure coding practices, injection vulnerabilities, or improper access control mechanisms. By analyzing the code before deployment, many vulnerabilities can be detected and mitigated early in the development process.

Dynamic code testing involves executing the integration in a controlled environment to observe its behavior and interactions. This testing can reveal vulnerabilities that are only apparent during runtime, such as insecure data transmission or improper input validation. By subjecting integrations to dynamic testing, potential vulnerabilities that may not be evident in static analysis can be identified and addressed.

Additionally, the vulnerability scanning mechanism can leverage databases that contain information about known vulnerabilities and security issues. By comparing integration code and configurations against these databases, the system can identify instances where integrations are utilizing outdated or insecure libraries, frameworks, or methodologies. This allows for timely updates and patches to be applied, minimizing the risk of exploitation.

To implement this scanning mechanism, a dedicated infrastructure and tooling need to be developed. This infrastructure would include an automated scanning system that integrates with the Home Assistant ecosystem, continuously monitoring and scanning newly added integrations. The scanning results would be reported to developers, allowing them to address any identified vulnerabilities promptly.

Furthermore, the integration scanning process can be complemented by establishing guidelines and best practices for developers to follow when creating integrations. These guidelines can include security recommendations, coding standards, and secure communication practices. By promoting secure development practices, the overall quality and security of Home Assistant integrations can be significantly improved.

By introducing a vulnerability scanning mechanism and supporting developers with security guidelines, the Home Assistant ecosystem can ensure a more secure and robust environment. Users can have greater confidence in the integrations they add, knowing that potential vulnerabilities are being actively identified and addressed. This proactive approach to security strengthens the overall resilience of the system and enhances the protection of user data and privacy.

Bibliography

[1] Eric Rescorla. *HTTP Over TLS*. RFC 2818. May 2000. DOI: 10.17487/RFC2818. URL: https://www.rfc-editor.org/info/rfc2818 (cit. on p. 8).

[2] Silvio Quincozes, Tubino Emilio, and Juliano Kazienko. «MQTT Protocol: Fundamentals, Tools and Future Directions». In: *IEEE Latin America Transactions* 17.09 (2019), pp. 1439–1448. DOI: 10.1109/TLA.2019.8931137 (cit. on p. 8).

[3] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246 (cit. on p. 8).

[4] Christopher W. Badenhop, Scott R. Graham, Benjamin W. Ramsey, Barry E. Mullins, and Logan O. Mailloux. «The Z-Wave routing protocol and its security implications». In: *Computers & Security* 68 (2017), pp. 112–129. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2017.04.004. URL: https://www.sciencedirect.com/science/article/pii/S0167404817300792 (cit. on p. 8).

[5] *Unifi Protect*. URL: https://www.home-assistant.io/integrations/unifi/ (visited on 07/13/2023) (cit. on p. 11).

[6] *IKEA's TRÅDFRI*. URL: https://www.home-assistant.io/integrations/tradfri/ (visited on 07/13/2023) (cit. on p. 11).

[7] *Google Assistant*. URL: https://www.home-assistant.io/integrations/google_assistant/ (visited on 07/13/2023) (cit. on p. 11).

[8] *Philips Hue*. URL: https://www.home-assistant.io/integrations/hue/ (visited on 07/13/2023) (cit. on p. 11).

[9] Eliot Lear, Ralph Droms, and Dan Romascanu. *Manufacturer Usage Description Specification*. RFC 8520. Mar. 2019. DOI: 10.17487/RFC8520. URL: https://www.rfc-editor.org/info/rfc8520 (cit. on pp. 13, 14, 19).

[10] Martin Björklund. *The YANG 1.1 Data Modeling Language*. RFC 7950. Aug. 2016. DOI: 10.17487/RFC7950. URL: https://www.rfc-editor.org/info/rfc7950 (cit. on p. 14).

[11] Ralph Droms. *Dynamic Host Configuration Protocol*. RFC 2131. Mar. 1997. DOI: `10.17487/RFC2131`. URL: `https://www.rfc-editor.org/info/rfc2131` (cit. on pp. 14, 19, 30).

[12] Suresh Krishnan, Siva Veerepalli, Eric Njedjou, Alper E. Yegin, and Nicolas Montavont. *Link-Layer Event Notifications for Detecting Network Attachments*. RFC 4957. Aug. 2007. DOI: `10.17487/RFC4957`. URL: `https://www.rfc-editor.org/info/rfc4957` (cit. on pp. 14, 19).

[13] Tim Berners-Lee, Larry M Masinter, and Mark P. McCahill. *Uniform Resource Locators (URL)*. RFC 1738. Dec. 1994. DOI: `10.17487/RFC1738`. URL: `https://www.rfc-editor.org/info/rfc1738` (cit. on p. 14).

[14] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. May 2008. DOI: `10.17487/RFC5280`. URL: `https://www.rfc-editor.org/info/rfc5280` (cit. on p. 14).

[15] Dimitrios Pendarakis, Dr. Raj Yavatkar, and Dr. Roch Guerin. *A Framework for Policy-based Admission Control*. RFC 2753. Jan. 2000. DOI: `10.17487/RFC2753`. URL: `https://www.rfc-editor.org/info/rfc2753` (cit. on p. 15).

[16] «IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity». In: *IEEE Std 802.1AR-2018 (Revision of IEEE Std 802.1AR-2009)* (2018), pp. 1–73. DOI: `10.1109/IEEESTD.2018.8423794` (cit. on p. 19).

[17] Michael Paik, Jerónimo Irazábal, Dennis Zimmer, Michele Meloni, and Valentin Padurean. *immudb: A Lightweight, Performant Immutable Database*. 2020 (cit. on p. 22).

[18] Pinyaphat Tasatanattakool and Chian Techapanupreeda. «Blockchain: Challenges and applications». In: *2018 International Conference on Information Networking (ICOIN)*. 2018, pp. 473–475. DOI: `10.1109/ICOIN.2018.8343163` (cit. on p. 22).

[19] Ali Sunyaev. «Distributed Ledger Technology». In: *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*. Cham: Springer International Publishing, 2020, pp. 265–299. ISBN: 978-3-030-34957-8. DOI: `10.1007/978-3-030-34957-8_9`. URL: `https://doi.org/10.1007/978-3-030-34957-8_9` (cit. on p. 22).

[20] Tony Hansen and Donald E. Eastlake 3rd. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234. May 2011. DOI: `10.17487/RFC6234`. URL: `https://www.rfc-editor.org/info/rfc6234` (cit. on pp. 23, 37).

[21] Fulvio Corno, Luca Mannella, et al. «A Gateway-based MUD Architecture to Enhance Smart Home Security». In: *Proceedings of the 8th International Conference on Smart and Sustainable Technologies–SpliTech 2023*. Institute of Electrical and Electronics Engineers (IEEE). 2023, pp. 1–6 (cit. on p. 27).

[22] Root Dir. «OpenWrt Development Guide». In: (2012) (cit. on p. 30).

[23] Kevin Yeich. «osMUD-Open Source MUD Manager». In: *Available on: https://github.com/osmud/osmud* (2019) (cit. on p. 30).

[24] *MUD Maker*. URL: `https://mudmaker.org/` (visited on 07/13/2023) (cit. on p. 30).

[25] *MUD Generator*. URL: `https://git.elite.polito.it/security/mud-generator` (visited on 07/13/2023) (cit. on p. 41).