Thin proxies: REST APIs

Contents

Apertis is a distribution that aims to provide solid bases to build products from IOT devices to complex HMI systems. The workflows to build such variety of products involves many different technologies, tools and developers'background.

A common issue during development is the need of high level APIs to interact with the system to allow application developers to focus on their use cases while hiding the complexity of low level system APIs.

Additionally, to hide complexity, a nice-to-have goal is to make these APIs technology agnostic, allowing developers to write their application in the language/technology they choose without any additional complexity.

An example of this situation is to let a developer with a basic network knowledge access a high level API to setup a simple network configuration in a HMI web application using Javascript. Unfortunately, currently, the only available solution to perform network configuration is to access the extremely complex **connman C D-BUS APIs**.

In order to provide a solution that satisfies these objectives, Apertis encourages the use of Thin Proxies.

Current status

System APIs

System APIs are used to allow application to interact with the system in order to:

- Retrieve information, such as retrieving network configuration
- Configure parameters, such as configuring network parameters
- Perform actions, such as system reboot

These kind of APIs are usually available in C language and bindings for other languages are built on top them, such as Python, Go and Rust. In the case of C++, usually the approach is to use the standard C API.

Since these kind of APIs are meant to cover many different use cases, they usually provide low level functionality, making them extremely big and complex. In addition they are very tied to specific technologies, requiring a deep knowledge in order to properly use them.

Lastly, as it is clearly seen, the use of this kind of API imposes security risks which should be minimized to provide a robust and reliable solution.

As a summary the challenges are:

- Usually built in the C language
- Provide low level functionality
- Very big and complex
- Tied to specific technologies
- Impose security risks

Application development

There is a wide range of applications which require access to system APIs to fulfill their goals. However, it is very common to only need to use a small number of high level operations. In such cases, accessing low level system APIs as described previously represents a huge barrier for development due to:

- Big learning curve for system APIs
- Big learning curve for technologies involved
- No up to date binding for the language of preference
- Error prone due to limited experience

Additionally, it is common to build Flatpak applications, in order to provide an easy way to distribute and upgrade confined applications, improving the security and robustness of the solution. Under these premises, using system APIs directly from Flatpak is not natural since it goes against the principle of application confinement. To solve that Flatpak applications usually communicate with the system services via D-Bus, but in some cases this is not ideal given it may still require low-level knowledge of the components in question and is tied to a specific IPC mechanism.

In these use cases a different approach should be to provide:

- Easy to use APIs
- Support for different languages/technologies
- Allow access from confined applications

Proposal

As a solution to overcome these difficulties Apertis encourages the use of Thin Proxies, to provide easy to use high level APIs for system APIs. The idea behind this concept consists in building a small service which provides an API targeted to the specific use case to provide the required functionality. This service should use REST APIs to provide a technology agnostic API.

There are other possible approaches, such as an IPC API which is used in other projects. However, a REST API is much more technology agnostic, allowing developers without experience in Linux systems to easily build applications.

REST APIs

A [REST API] uses HTTP to access resources using the standard methods GET, PUT, POST and DELETE. This type of API is based in the concept of representational state transfer (REST), with aims to allow scalability. The goals behind using these kind of APIs are:

- Improve scalability of interactions between components
- Stateless operations
- Uniform interfaces
- Independent deployment of components
- Facilitate caching
- Enforce security
- Support layered system

For these reasons REST APIs are the most common technology to provide access to remote resources on the Internet, allowing developers to build applications in the language they prefer.

This same approach can be used to build applications that access local resources, such as system APIs, using similar workflows than the ones used in other applications.

REST APIs design

Designing a REST API can be challenging since any change might impact in the clients that make use of it. In this context the following recommendations should help to reduce the possibility of having to deal with unexpected changes.

Analyze use cases

The first step in the process of designing an API is to understand what the requirements of the use cases are. This step should also try to think about possible new use cases with new requirements, to create an API that could be extended naturally in new versions.

Data format

REST APIs can used with different data such XML or JSON, however, nowadays is recommended to use JSON, since it is de-facto format for sending and receiving data.

URIs and endpoints

Connected to the previous comment, designing URIs and endpoints, considering current and possible future uses cases will help developers using the REST API when developing their applications and supporting it across new versions of the API.

Additionally, when choosing names for endpoints it is recommended to use *nouns*, since they represent objects, while the action on those objects is represented by the HTTP method used. In relation to this, make use of logical nesting on endpoint to show relationships between them, making the API easier to use.

Error handling

In order to make the API easier to use, *errors* should be handled gracefully and standard HTTP codes, alongside additional text to describe the error, should be be returned when needed.

Versioning

Using versioning in the API ensures that the evolution of an API does not affect old unsupported clients which are tied to an old version of the API while allowing well supported clients to have access to the latest functionality.

Documentation

The process of documentation is important since this will allow to share the design with the people involved, which will provide valuable feedback regarding missing functionality or possible issues.

REST APIs security

As previously mentioned, one of the goals of this proposal is to provide a solution that helps developers reduce the overhead without sacrificing security.

To do so, the same security principle that apply to any REST API on the Internet also applies to Thin Proxies. These security principles can be relaxed during development and testing but should be enforced in production.

Block not allowed HTTP methods

A common premise in security is to only allow the really needed functionality, in order to reduce the possibility of exploits and attacks. With that in mind only the HTTP methods that should be supported should be whitelisted, while other methods should be blocked. As an example, the HTTP method DELETE is usually not supposed to be used on common API calls and should be blocked.

Use TLS and well supported security framework

Nowadays, TLS is widely used on the Internet since it is the base for any secure service. A very good practice is to make use of a well supported security framework since this enables updates and security fixes to make the application, in this case the Thin Proxy, more robust. By using TLS on REST APIs, both confidentiality and server/service authentication are supported.

During development and testing the use of encrypted channels can be disabled to help developers to debug their applications.

Use solid authentication mechanism

One of the most challenging aspects of security with REST APIs is to have a solid authentication mechanism to prevent unauthorized access to the services provided by the application.

The following are recommended mechanisms:

- API keys[1]: solution based in the generation of a secret for the user, which should be stored in the server/service.
- OAuth[2]: solution to pass authentication information from one service to other, which offloads the authentication to a different service.
- JSON Web Tokens (JWT)[3]: solution based on token generation from JSON data which are signed by the service.

Data in URL

It is discouraged to use sensitive data in the URL since this could lead to a security leak. Data such as usernames, passwords and tokens should not be included in the URLs since they could be easily captured by the server/service logs or a network sniffer for example.

Validate input parameters

One way in which security can be compromised is by assuming that input data is valid, since processing invalid data could lead to unexpected results. This can happen either by an error in the application which is using the API or by malicious users who are trying to get access to system resources.

By validating the input data the system will be able to reject API calls that use invalid data minimizing the possibility of issues.

Include timestamps

Adding timestamps to HTTP request allows the server/service to check them against the current time and in this way filter old requests preventing reply attacks. This also adds value information to log records which could help during debugging.

[1] https://en.wikipedia.org/wiki/Application_programming_interface_key
[2] https://en.wikipedia.org/wiki/OAuth
[3] https://en.wikipedia.org/wiki/JSON_Web_Token

Log failed requests

Understanding what is happening is a key element to ensure security, and to quickly respond to a security breach. In this context, having information related to failed requests or abnormal situations, provides the data required to analyze potential security issues.

REST APIs security for local services on embedded devices

The use of REST APIs is widely used to access services on the Internet but as mentioned is this document the same principles can be used to access local services in embedded devices. In this context, the same security principles apply but some important differences need to be noted.

Block requests from other hosts

The fact that both client and server are in the same host allows to add additionally restrictions, such as only allowing requests from local host. This restriction could be relaxed on development environments to allow developers to easily test their code.

Solid authentication mechanism for embedded devices

This document has covered some authentication mechanisms that are used on the Internet, however not all of them are suitable for this use case. From the previous list, JWT is the most suitable for this scenario, since the token is signed with the service's private key and contains all the required information in the token itself. In this way, the server/service does not need to know in advance which clients will use it and does not need to store specific data for the client application, which simplifies client deployments and avoid requiring a database.

Consuming REST APIs

The idea behind this document is to enable developers to consume the REST APIs provided by Thin Proxies from the language/technology of their preference. In this regard, only high level suggestions can be made that are tied to the recommendations already suggested.

Use TLS and well supported security framework

As mentioned in the previous section, the use of TLS and well supported security frameworks improves the security of the solution, since an attacker could make use of a vulnerability in the consumer code to gain access to the system.

As commented before, during development and testing the use of encryption can be disabled to help developers to debug their applications.

Sensitive data

Sensitive data such as username, JWT tokens or any other data that could give useful information to attackers should be stored encrypted to minimize security

risks. Additionally, following the same principles, this type of data should not be included in trace logs since it could be easily retrieved.

Conclusion

The use of Thin Proxies based on REST APIs provides a way for developers to build their application using the language/technology they prefer, hiding the complexity of low level APIs.

Links

- REST API Best Practices[4]
- How to create a RESTful API[5]
- Securing REST APIs[6]
- How to secure REST API endpoints[7]
- REST API web service security[8]

[4] https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/
[5] https://yalantis.com/blog/how-to-create-a-restful-api/
[6] https://developer.okta.com/blog/2019/09/04/securing-rest-apis
[7] https://www.techtarget.com/searchcloudcomputing/tip/How-to-secure-REST-API-endpoints-for-cloud-applications
[8] https://www.netsparker.com/blog/web-security/rest-api-web-service-security/