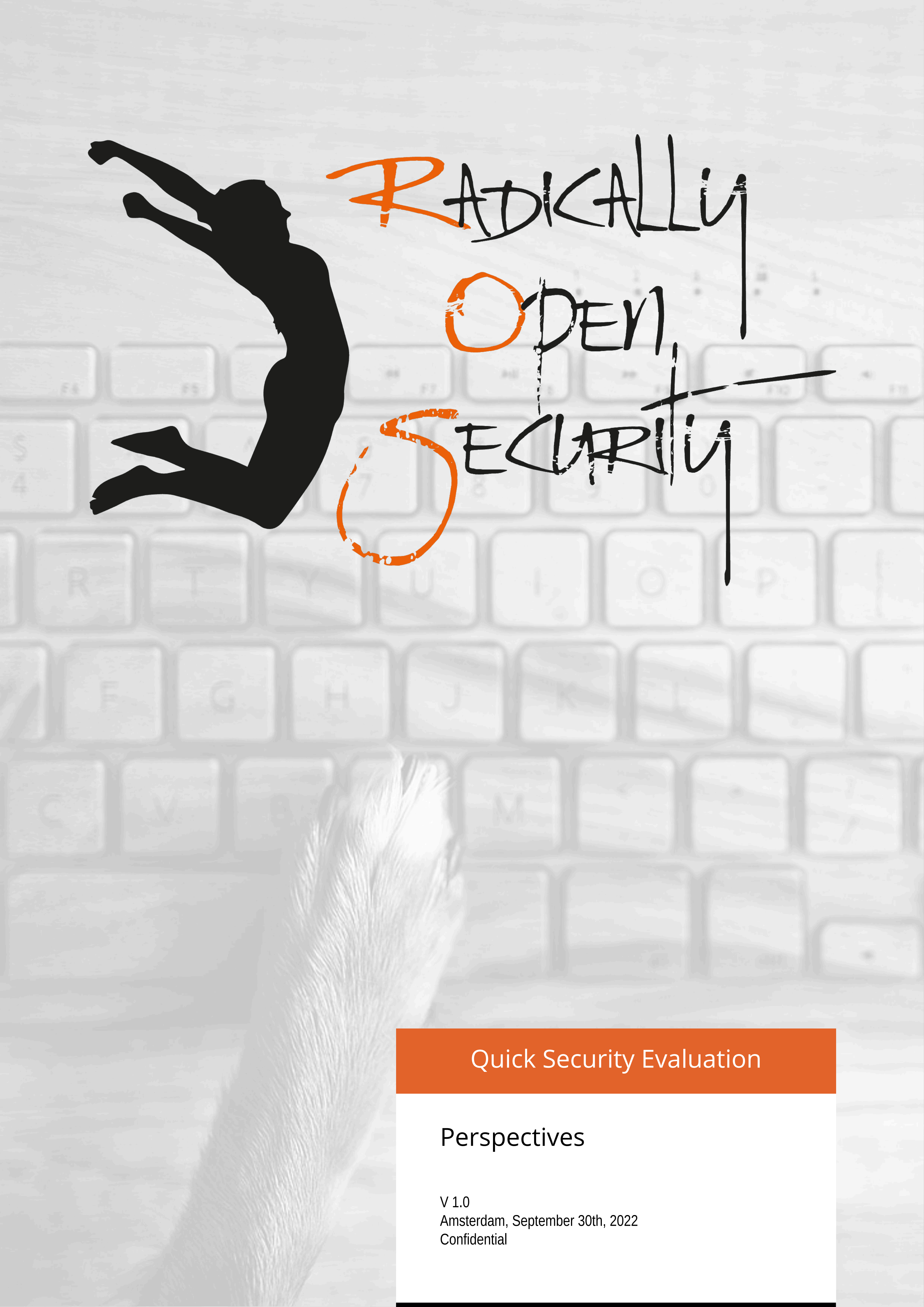

Quick Security Evaluation

Perspectives

V 1.0
Amsterdam, September 30th, 2022
Confidential

1 Introduction

As part of the NLnet projects your project **Perspectives** has received a basic quick security evaluation from Radically Open Security. The goal of the review is to provide advice and input to consider in the further development of your project. The selected project gets 4 persondays from ROS for a quick security evaluation.

2 Security Evaluation For Perspectives

Tasks Performed

- Discussion with representatives of the project via messaging and via video conference.
- Analysis of the document *A_Security_Perspective_on_the_Distributed_Runtime_v3.pdf* provided in the ROS git repository *off-ngid-perspectives2022*.
- Analysis of the document *A_Security_Perspective_on_the_Distributed_Runtime_v4.pdf* provided in the ROS git repository *off-ngid-perspectives2022*.
- Analysis of the *Security Concerns* section in the two aforementioned documents.
- Analysis of *Getting started with Perspectives* accessible at Getting started with Perspectives.
- Quick analysis of the InPlace application accessible at https://inplace.works/.
- Quick analysis of the broker service by the InPlace application to send and receive chats messages.
- Quick analysis of the exposed infrastructure (e.g., RabbitMQ, Web Server).
- Analysis of the authentication and authorization model of the application.
- Analysis of the cryptographic assumptions in the application.

Security Considerations

The following section contains a series of security consideration which, if adopted, would help to improve the security posture of the project.

Devise a Formalized Threat Model

The system lacks a formalized Threat Model and relative security requirements which could guide the project development. Citing the Wikipedia page for Threat Modeling:

Threat modeling is a process by which potential threats, such as structural vulnerabilities or the absence of appropriate safeguards, can be identified and enumerated, and countermeasures prioritized.[1] The purpose of threat modeling is to provide defenders with a systematic analysis of what controls or defenses need to be included, given the nature of the system, the probable attacker's profile, the most likely attack vectors, and the assets most desired by an attacker. Threat modeling answers questions like “Where am I most vulnerable to attack?”, “What are the most relevant threats?”, and “What do I need to do to safeguard against these threats?”.

A threat modeling exercise will help the project to understand the major weak points of the system and its most critical component.

Additionally, the output of a threat model will include a series of security requirements which, similarly to functional one, will guide the development phases and help to secure the system.

Ensure Uniqueness of the GUID Identifying the User

Each user on the system is identified by a *GUID* generated during the account creation. Such GUID will be then used to identify the user and its actions.
This approach relies on the assumption that two users will never have the same GUID. This is a problem in single system deployment and can become a huge risk in distributed ones.
Imagine this scenario in a distributed system:

- Alice creates an account on System1.
- Alice gets assigned the GUID *0e460b23-cf02-427a-b081-1c929f4894bc*.
- Bob creates an account on System2.
- Alice gets assigned the GUID *0e460b23-cf02-427a-b081-1c929f4894bc*.

Now Alice and Bob have the same GUID, and it will not be possible to distinguish between the two and inter-system communication would be difficult if not impossible.
In single system deployment the issue is still present with the difference that a collision is less likely. In this case the recommendation would be two:

- On user creation, verify the uniqueness of the newly created GUID.
- On distributed system devise a mechanism to disambiguate collisions.

One solution for distributed system could be the definition a hierarchical organization for the deployments with the priority on the GUID on the highest one in the chain. For example, define a root deployment which GUID, in case of collision, will have the precedence with respect to other and newer deployment.

Password Protect the User Account

In the current configuration, the application does not require the user to provide a password to identify themselves but relies only on their username.
This approach risks exposing users accounts to malicious actors since the username is rarely a secret. Additionally, due to the collaborative nature of the application, the username is guaranteed to not be a secret.
The recommendation would be to implement an authentication mechanism requiring, at least, a username and a password. A good starting point to study authentication mechanism is the Authentication Cheat Sheet from the *OWASP Cheat Sheet Series*.

Securely Store Users' Passwords

Once the user will be able to set their password a new problem will arise: how to securely store them. Once again the OWASP project can help in understanding the problem and relative solution with its Password Storage Cheat Sheet.
The recommendation would be to store the password using a secure hashing algorithm, as for example *Argon2id*, or a key derivation function, like *PBKDF2*, combining it with a salt and possibly a pepper.

Signing Keys Management

In the future the transaction will be signed by the user. These requirements implies that the backend will have some information to verify such signature. One solution would be to have a copy of the *public key* of the user to verify the signature. Now the question arises, how to associate a user to its public key?

A possible solution would be to ask for the user's public key on registration. Any subsequent requests for changing the current public key will require a signed request, which will provide a proof that the user has possession of the private key. Lastly, since the system will have to verify the signature for each message, it is recommended to store the public key of the user locally. This to minimize the possibility for an attacker to perform a Man in the Middle and swap the key in transit.

Encrypt Delta Messages

In the document *A_Security_Perspective_on_the_Distributed_Runtime_v4.pdf* it is stated that the deltas will be encrypted even if, at the moment of writing this document they are not.
The encryption of the deltas the user and the system will need to have a shared key, if the systems decides to use a symmetric algorithm or, or the user will have to have a private key provided by the server.
In case a symmetric algorithm is chosen server and client will have to exchange a key. A solution to this problem could be to implement a Diffie-Hellman key exchange. This algorithm for exchanging a key has the advantage that the key is never transmitted and so, even if an attacker were to intercept the traffic it would be impossible to recover the key.
An easier solution would be for the server to provide the clients with its public key to encrypt the deltas. In this way each client will be able to secure the content but no other one will be able to decrypt it, with the exception of the server.
While the later solution has the advantage of simplicity, the first one has the advantage of providing forward secrecy.
Which one to chose depends on the kind of threat actor the system expects.

Encryption of Information in IndexDB

When the user creates an account they have the possibility to use *IndexDB* to store information. This configuration exposes users data in case of shared workstations. For example, imagine a user wanting to use the application in an internet cafe', all their data will be exposed to the next person using the workstation.
One solution to resolve the issue could be to encrypt the data in the *IndexDB* with a password different for each user.
Such password could be derived using a key derivation function (e.g., PBKDF2) and should be never stored.
Since the frontend is developed using react, it could be possible to use a middleware to encrypt and decrypt the information transparently from the point of view of the application.

Manage Third Party Dependencies (Supply Chain)

The application, as every system build today, relies on external libraries to implement its functionalities. While this is normal and perfectly acceptable, every time a new dependency it is introduced the attack surface of the system increases since any vulnerability in any dependency will become a vulnerability in the application itself.
Since it is not possible to know preemptively the vulnerability which will arise, it becomes necessary to closely monitor our dependency and their issues. This task is achievable by using third party tools.
For example, to track issues in dependencies there are two notable tools:

- Dependabot
- Snyk.io

The first one is a service directly from GitHub and is able to generate alerts (when properly configured) when known vulnerability are discovered Dependabot page.
The second solution, namely Snyk.io is a commercial solution with auto Pull Requests capabilities.

Window.postMessage considerations

One of the security concerns of the project, which ROS was explicitly asked to explore, is about the safe usage of *Window.postMessage* and its counterpart *window.addEventListener*.
A primer on the safe usage of such methods can be found in the public Mozilla documentation about Window.postMessage().
While reading the documentation is highly advised, the general recommendation to safely use the methods are:

- When using *Window.postMessage* clearly specify the *targetOrigin* parameter (i.e., https://inplace.works).
- In the handler for *window.addEventListener* clearly validate the event origin (i.e., https://inplace.works).

Recommendations

- Consider creating formalized Threat Models for the system or at least for the main components
- Consider implementing a strategy to manage conflicts in GUID
- Consider password protecting the user account
- Consider securing the user's password using a well known hashing algorithm using salt and pepper
- Consider implementing a mechanism to associate an identity to a public key
- Consider encrypting the Delta Message
- Consider encrypting the information stored in IndexDB
- Consider implementing monitoring for the projects dependencies
- Consider specifying the targetOrigin and verifying it when using Window.postMessage and window.addEventListener

3 Contact details

Your pentester(s) for this project:

- Jacopo Ferrigno
  Jacopo Ferrigno holds a M.Sc. degree from Politecnico di Milano and routinely participates to capture the flag competition. He graduated with a thesis on binary analysis techniques based on the callgraphs. He is specialized in reverse engineering and hardware analysis.

If you have any questions about this advice, please contact us at info@radicallyopensecurity.com

For more information about Radically Open Security and its services please visit our website: www.radicallyopensecurity.com.

4 Disclaimer

This evaluation is not to be considered a full audit or pentest. It is important to understand the limits of ROS' services. ROS does not (and cannot) give guarantees that something is secure. Additionally, the above advice is obviously incomparable to a full-blown security audit as performed by ROS or any other professional security company, which takes orders of magnitude more time and effort. (Such an audit may be necessary in the future still, and we would be happy to work with you on that).

We recommend for now you treat our feedback as a discreet sanity check by knowledgeable friends; it is not the intention to publicly claim that Radically Open Security audited your code and found it to be safe or 'found no problems'. Rather, the intention is the reverse: we aim to help you capture obvious flaws within the very limited terms of this deal, and to protect the general audience from irresponsible projects putting them at risk. NGI Zero takes the public interest very seriously and wants to have at least a rough understanding of the maturity of your work - hopefully this will guide your own behavior in terms of any claims you make.

There is no shame in clearly messaging to users that your project is still in an early stage, that they use it at their own risk and that there are no guarantees - being honest with your users can only ever be a good thing.