Inertia Analysis Final Report

CPEN 442

Natthan Leong
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
natthan@alumni.ubc.ca

Lucy Zhao
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
lucyzhao.zlx@gmail.com

Mason Duan
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
masonkbduan@gmail.com

Stefan Jauca
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
stefan.jauca@gmail.com

***Abstract*—Inertia is a tool for simplifying the set up and management of software projects on virtual private servers. Disruption to Inertia negatively affects software development processes and can incur undesired financial costs. Analysis of Inertia was performed using static analysis tools, code review, and through manual testing and usage. We identified an abuse case with failed login attempts and account deletions that could be exploited by a network user. We discovered that 10 incorrect login attempts would lead to the automatic deletion of the user account and that any logged-in user may view all other usernames for use in mass account deletions.**

I. Introduction

Inertia is an application that enables quick setup and management of continuous project deployments on any virtual private server (VPS) provider. It is developed by UBC LaunchPad, a student-run software engineering team. It aims to help students deploy their code projects with minimal friction and to easily switch between VPS providers.

As the Internet continues to grow, so does the number of server side web applications. According to edgescan's 2018 report, web application security is still the area of most risk from a security breach [1]. Edgescan estimates that the 19.2% of the risks discovered for internet-facing applications were high and critical. As Inertia is an application that enables continuous project deployments on any virtual private server, it is also susceptible to these risks.

The application consists of two main components: a command line interface and a daemon running on the remote server.

No external security analysis had been previously done on Inertia, however the system's developers have catalogued several known vulnerabilities as comments in the source code. One similar system that has been analyzed from a security perspective is Heroku, which informed our analysis. Inertia also relies on 3rd party APIs and services which have been analyzed in the past. This available literature was reviewed as part of our analysis of Inertia.

The system was analyzed through actual usage, code review, and with the help of static analysis tools. Abuse of account deletions upon incorrect login attempts was discovered. Non-administrator users are able to obtain all account usernames which can lead to further account deletions. We recommend that account locking with login attempts monitoring be implemented and that the ability to view all usernames be reserved for administrator accounts.

Through our analysis project, we identified vulnerabilities present in Inertia as summarized above.

II. Analyzed System

Inertia consists of five major components: the command line (CLI) application on the user's computer, the remote (VPS), the daemon running on the VPS that builds the Docker project, the Docker project itself and a code repository. There are four major communication links: CLI $\rightleftharpoons$ remote server, CLI $\rightleftharpoons$ daemon, daemon $\rightleftharpoons$ GitHub repository, and daemon $\rightleftharpoons$ Docker project. The project contains 11353 lines of Go code as measured by gocloc [2] for Inertia v0.5.3 [3].

The CLI is an application written in Go that provides an interface for managing the VPS and the Inertia daemon. The CLI also handles all local configuration. The HTTP request from the user's CLI consists of commands that start, stop, and status retrieval of the deployed project. The CLI initiates the Inertia setup by executing a script over SSH that installs Docker, pulls and runs a `ubclaunchpad/inertia` image from DockerHub, and sets up an additional RSA key and JSON Web Token for authentication.

The remote VPS, such as a Google Cloud Compute or an AWS Elastic Cloud Compute (EC2) instance, hosts the Inertia daemon and the user's Docker project.

The daemon, also written in Go, runs on the VPS and waits for commands. The commands can be either HTTP requests from the CLI or WebHook events from GitHub. Depending on

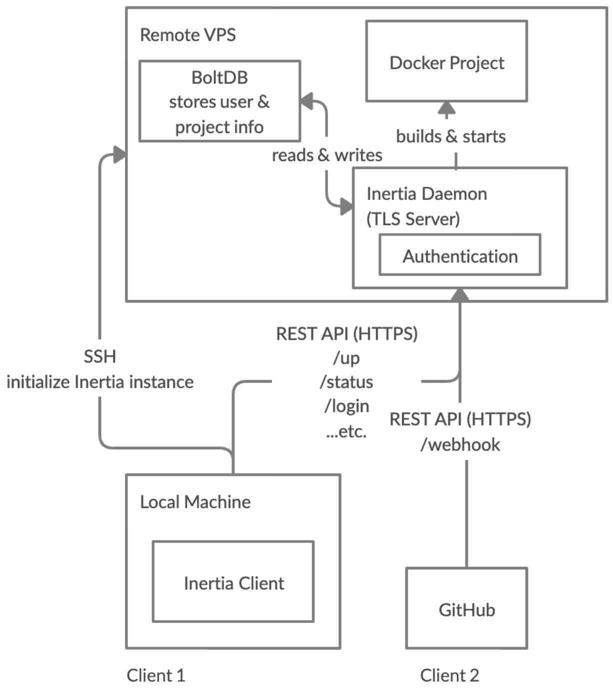

Remote VPS
BoltDB stores user & project info
Docker Project
builds & starts
reads & writes
Inertia Daemon (TLS Server)
Authentication
SSH initialize Inertia instance
REST API (HTTPS) /up /status /login ...etc.
REST API (HTTPS) /webhook
Local Machine
Inertia Client
GitHub
Client 1
Client 2

Fig. 1. Inertia project architecture adapted from [3].

the command it receives, it can build, stop, or start the Docker project.

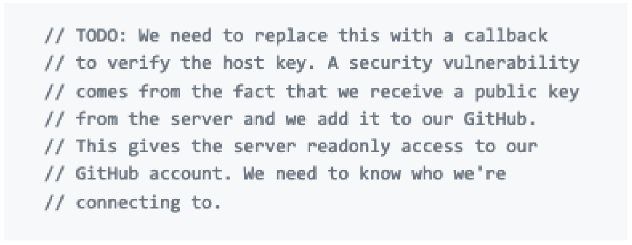
Docker runs on the user's remote host and provides a common environment, i.e., a container, for the user's application to run.

The GitHub repository contains the the source code for the project. When the source code changes, WebHooks are sent from GitHub to the Inertia daemon such that Inertia can automatically make updates to the project.

III. Related Work

We reviewed internal design documents produced by UBC LaunchPad including comments in their source code indicating potential vulnerabilities that were identified already by their developers. For example, the Inertia client code makes use of SSH host keys for connecting the user's server to GitHub, giving read-only access to the user's repositories. However, at the time of writing, their implementations use self-signed certificates (as opposed to certificates signed by a trusted third party) and as such, the certificate is not verified by the Inertia client. This means that all incoming requests result in the server being granted read-access to the GitHub repositories, with no meaningful verification of the authentication of the server. Although this security vulnerability is present in the analyzed version of the Inertia client, it is appropriately identified in their source code documentation, as seen in Fig 2.

Security analysis on a similar system, Heroku, was done by freelancer Charlie Egan [9]. This was the most compre-



```
// TODO: We need to replace this with a callback
// to verify the host key. A security vulnerability
// comes from the fact that we receive a public key
// from the server and we add it to our GitHub.
// This gives the server readonly access to our
// GitHub account. We need to know who we're
// connecting to.
```

Fig. 2. Security vulnerability already identified in source code [4].

hensive study we found on a system that greatly resembled Inertia. Egan's analysis revealed certain common areas that vulnerabilities may exist, such as DDoS attacks, inadequate firewall configurations, ease of social engineering exploits, and the usage of cookies for session authentication.

Several dependencies used in Inertia have also been subject to external security analyses. Usage of self-signed SSL certificates is known to be insecure, as documented by Pornin [11], and usage of cookies as a means of session authentication has security vulnerabilities according to the InfoSec Institute [13]. GitHub WebHooks are used by the Inertia daemon, and they rely on the SHA1 algorithm [14], which has not been considered secure since as early as 2005 [15].

IV. Analysis Methodology

A. System Analysis

Our analysis methodology consisted of three main approaches: static analysis, code review, and manual testing.

First, we performed static analysis on the source code of both the CLI and the daemon using two tools: Golang Security Checker (gosec) and staticcheck. Gosec scans the abstract syntax tree of the source code specifically for security problems [7], whereas staticcheck is a non-security static analysis tool for detecting programming logic errors [8].

Through a divide and conquer strategy, code review was performed by splitting analysis of the various components amongst team members. Analysis of the HTTP API endpoints were performed by code inspection. During code review, we focused on the usage of cryptography and the communication between various components of the system, such as CLI to daemon, CLI to VPS, daemon to GitHub, and daemon to Docker.

For analysis through actual usage, we deployed a NodeJS application [5] on an Amazon EC2 [6] instance. We hosted our code on GitHub and used the Inertia CLI from our local development machines. With this environment, we performed a number of operations such as creating projects, creating users, adding users, getting new users to set up the project on their own machine, and bringing the server up and down. We monitored the state of Inertia's database as we attempted to uncover abuse cases. Drawing on the indicators uncovered by our code review, we followed leads to see if we could replicate the attacks that were hypothesized by code inspection. This approach of identifying potential vulnerabilities via code analysis and subsequently demonstrating exploits on our own machines was particularly fruitful. Another dimension of our manual testing involved using Wireshark to sniff communication packets, looking for any weaknesses or exposed data in network communications.

B. Ethical Considerations

The users of the Inertia system were at the forefront of our minds as we undertook this analysis. Our motivation is to uncover vulnerabilities so that the users of the system could use Inertia with less worry for the security of their applications under development. We did not want the system's users or developers to be adversely affected in any way as we undertook our analysis.

The approach and methodology taken by our team deliberately aimed at reducing the chance of any adverse affect on both developers that use Inertia and the developers of Inertia themselves. Inertia is an open-source project, hence there was no danger of compromising confidentiality of the source code. We only performed manual testing on instances of Inertia running on our own machines and servers, entirely avoiding the risk of disrupting the availability of Inertia or of others' server and applications, for other users who have their own live instances of Inertia. Finally, we did not make any alterations to the code hosted on GitHub, which can risk compromising the integrity of the system. Another benefit of this methodology is that it prevented premature disclosure of any findings as we were not deliberately or accidentally making any vulnerabilities public.

C. Risk Management

In order to mitigate the risks associated with our analysis, we obtained written permission including a legal consent form from the system owners that explicitly authorizes us to analyze the security of the Inertia system, and includes our agreement upon the terms for responsible disclosure.

We analyzed Inertia's use of 3rd party APIs and services only to the extent of how Inertia interfaced and implemented them. As such, we restrained ourselves from causing unintended harm to others in tampering with systems beyond the scope of what we had obtained consent for. This protects us from the risk of breaching the terms of agreement of any of those 3rd party services APIs.

V. Results

A. Authentication System Vulnerabilities

TABLE I
Inertia Access Token Types

Token Type	Duration of Validity	Permissions
User Token	2 hours	Get deployment status and logs; List all users
Admin Token	2 hours	Make new deployments; Add/remove users
Master Token	Does not expire	Same as admin token

Inertia uses JSON Web Tokens (JWT) for authentication. When a user logs in, the daemon returns a JWT to be stored in the user's local file system. This token is then passed to the daemon through HTTPS for subsequent API calls. Through code review, we summarized the three types of JWTs present in the Inertia system. As can be seen in Table 1, Inertia employs a master token, which does not have an expiry date. The master token allows for a non-expiring CLI session for individual Inertia users. Having a non-expiry master token could introduce difficulties to the secret management process if Inertia is used by a team.

Through code review, we discovered that 10 failed login attempts will cause a user's account to be deleted [16] by design. Through manual testing, we discovered that a non-administrator user can also list all usernames for an Inertia instance [17].

B. Vulnerable Dependencies

An analysis of dependencies revealed three vulnerable dependencies in Inertia v0.5.3 with which should be resolved as soon as possible: Inertia currently uses gorilla/websocket v1.4.0, which has a high severity security advisory [18] regarding a potential Denial-of-Service (DoS) vector; Inertia currently uses microsoft/go-winio v0.4.12, which has a race condition that is fixed in v0.4.13 [19]; dgrijalva/jwt-go, the epoch time expiry passes as valid remains unfixed [20].

VI. Discussion

A. Interpretation of Results

Implementing a master token with no expiry date could introduce difficulties to the secret management process if Inertia is used by a team. Once a team member leaves, the team has to rotate the key used for JWT signing to create a master token with a different signature. This could be a costly process because it invalidates all other access tokens.

The 10 failed login attempts can be abused to delete accounts. Without prior knowledge of other usernames, an attacker can easily retrieve all other usernames through a compromised account.

B. Adversary Model

We considered two main adversary models. First, we took the case of a disgruntled team member. This individual may desire to disrupt the work of a particular teammate or perhaps seek to undermine the project as a whole. He or she has access to the usernames of fellow developers, and is a non-admin user in Inertia. Furthermore, he or she knows the IP address of the team's VPS instance. Thus, it is fairly trivial for this adversary to disrupt the work of a particular user (by repeatedly deleting them via 10 consecutive invalid login attempts) and by extension, since the adversary can list all users, he or she could perform the attack of deleting all Inertia users with this information and abilities.

Secondly, we considered a network user. This user could be motivated to disrupt the work of a team of developers using Inertia. Perhaps they would like to conduct a form of ransom campaign by blackmailing teams using Inertia, or they are seeking more generally to compromise instances of virtual private servers. This adversary may not be a single individual, but could be a corporate entity, such as a rival team of developers. A network adversary can perform web crawling and port scanning to detect if an Inertia daemon instance is listening for commands at a particular IP address. As such, it can be assumed that a network adversary also has access to the IP address of the victimized team's VPS instance. This information is sufficient for a network user to compromise user accounts through multiple login attempts, since they can most likely stumble upon a username either by trying likely possibilities (lowercase names, similar to a dictionary attack) or by learning which team of developers owns that IP address and performing an intelligent series of guesses based on this detective work. Once a single username has been revealed and the associated password guessed correctly, the attack of deleting all other existing users can be accomplished easily.

C. Principles of Designing Secure Systems

By violating fail-safe defaults in deleting user accounts instead of locking user accounts, the developers of Inertia allowed for abuse of a security measure that was meant to prevent account compromise.

By not adhering to the principle of least privilege, the developers of Inertia allowed non-administrator users, which may be controlled by compromised accounts, to have access to the full list of administrator and non-administrator usernames.

By not adhering to the monitor and trace principle, the users of Inertia cannot determine who caused the deletion of users on an Inertia instance as there are no logs regarding the origin of the repeated login attempts.

VII. Recommendations

a) Rate limit failed login attempts: We recommend that instead of deleting the user account after 10 unsuccessful logins, the authentication system limits login attempts from the same account to 1 per 5 seconds, with a 1 hour account lockout after 10 attempts. This follows the principle of fail-safe defaults because when login fails for multiple times, it defaults to the safe outcome of delaying the next login. This ensures that the legitimate user can still access his/her account after the lockout period, at the same time it prevents brute-force password attacks. We chose this recommendation because it maintains the availability of the system for legitimate users, while achieving the same effect of preventing brute-force attacks as the existing measure (user account deletion). According to the "Usable Security" lecture on "Usable Authentication and Passwords", a relatively secure password, such as "alpine fun", takes more than 1000 years to crack with the above password retry scheme. Therefore, this recommendation is both practical and effective.

b) Return all usernames only if requested by an administrator user account: We recommend that usernames can only be listed by an administrator. This follows the principle of least privilege, because regular users do not need the ability to list all users in the system to interact with Inertia. This recommendation is simple to follow because it only requires changing the permission level of the "list user" API endpoint. It maintains Inertia's usability because the user list can still be obtained from an administrator account.

c) Log failed login attempts: We recommend that information pertaining to the failed login attempts be logged with information detailing the time of login, the originating IP address(es), and the username of the account being logged into. This follows the principle of monitor and trace because the details of the failed attempts would be logged and can be used for monitoring. We chose this recommendation because it helps in locating the attacker and hopefully deter future attacks. It maintains Inertia's usability as it would not impact legitimate users at all.

d) Disable master token when Inertia is shared by a team: We recommend that when shared access to an Inertia instance is enabled, Inertia makes it mandatory to create an admin account, and disable the master token. Since permanent access to Inertia is not permitted in a team setting, there is no need for non-expiring tokens. As can be seen from Table 1, the only difference between a master token and an admin token is the duration of validity. Therefore, the functionalities of a master token can be completely replaced by admin tokens. This follows the principle of least privilege because it prevents those who have left the team from accessing the system through an access token that they previously had.

e) Update dependencies to most secure version available: We recommend that Inertia's dependencies to be patched as soon there as there is a more secure version made available. This follows the principle of defense in depth as to be proactive about the security of the program. It maintains Inertia's usability as it would not impact legitimate users at all.

VIII. Conclusion

Inertia streamlines the process of web application development by allowing teams to interact more simply with their VPS and manages team member access. The methodology of our team in analyzing Inertia involved static analysis tools, code review, and manual testing. Four key vulnerabilities present in Inertia are: (1) the unsafe handling of repeated failed login attempts by deleting the account in question after 10 attempts, (2) the violation of the principle of least privilege by allowing any user to list the usernames of all users, and (3) allowing previous team members to access the system through a non-expiring token, and (4) the lack of any monitor and trace scheme that would allow administrators to audit the actions of users and allow them to detect attacks after they have occurred. We demonstrated a class of attacks due to these vulnerabilities that could easily be undertaken by a disgruntled team member and that could reasonably be exploited as well by a network adversary. The aforementioned vulnerabilities in Inertia can result in major workflow disruption by compromising the availability of Inertia for the entire team of developers. To address each vulnerability, we recommend applying secure design principles in the following form: (1) moving from account deletion to account locking, (2) restricting the capability of listing all users to only Inertia project administrators, (3) implementing a straightforward log of user login activity to facilitate post-attack investigations, (4) disabling non-expiring master tokens for team mode, and (5) patching vulnerable dependencies. Adopting our recommendations will result in a more robust defense against attacks seeking to disrupt the workflow of web application developers using Inertia.

We would like to acknowledge and thank UBC LaunchPad and the system owners of Inertia, for making this analysis project possible. We would also like to thank Larry Carson, Associate Director for Information Security Management at UBC, for his input during our presentation at the CPEN 442 Mini Conference.

References

[1] "2019 VULNERABILITY STATISTICS REPORT," *edgescan.com*, 2019. [Online]. Available: www.edgescan.com/wp-content/uploads/2019/02/edgescan-Vulnerability-Stats-Report-2019.pdf. [Accessed: 12-Nov-2019].

[2] hhatto/gocloc, "hhatto/gocloc: A little fast cloc(Count Lines Of Code)", *GitHub*, Available: https://github.com/hhatto/gocloc, 2019. [Online]. [Accessed 28-Nov-2019].

[3] UBC Launch Pad, "ubclaunchpad/inertia at v0.5.3," *GitHub*, Available: https://github.com/ubclaunchpad/inertia/tree/v0.5.3, 2019. [Online]. [Accessed: 8-Oct-2019].

[4] UBC Launch Pad, "inertia/ssh.go at v0.5.3," *GitHub*, Available: https://github.com/ubclaunchpad/inertia/blob/v0.5.3/client/ssh.go#L175-L181, 2019. [Online]. [Accessed: 25-Oct-2019].

[5] "Dockerizing a Node.js web app," *Node.js.* [Online]. Available: nodejs.org/en/docs/guides/nodejs-docker-webapp/. [Accessed: 05-Nov-2019].

[6] "Amazon EC2," *Amazon.* [Online]. Available: aws.amazon.com/ec2/. [Accessed: 05-Nov-2019].

[7] "securego/gosec," *GitHub.* [Online]. Available: github.com/SecureGo/gosec. [Accessed: 05-Nov-2019].

[8] "dominikh/go-tools," *GitHub.* [Online]. Available: github.com/dominikh/go-tools/tree/master/cmd/staticcheck. [Accessed: 13-Nov-2019].

[9] C. Egan, "Security discussion of the Heroku platform," *charlieegan3.com*, Available: charlieegan3.com/projects/heroku_ra_2016.pdf, 2016. [Online]. [Accessed: 8-Oct-2019].

[10] J. Ellingwood, "An Introduction to Securing your Linux VPS," *digitalocean.com*, Available: www.digitalocean.com/community/tutorials/an-introduction-to-securing-your-linux-vps, March 4 2013. [Online]. [Accessed: 8-Oct-2019.

[11] T. Pornin, "What are the risks of self signing a certificate for SSL," *security.stackexchange.com*, Available: security.stackexchange.com/questions/8110/what-are-the-risks-of-self-signing-a-certificate-for-ssl, October 13 2011. [Online]. [Accessed: 8-Oct-2019].

[12] Y. Ryabova, "HTTPS doesn't mean safe," *www.kaspersky.com*, Available: www.kaspersky.com/blog/https-does-not-mean-safe/20725/, January 17 2018. [Online]. [Accessed: 8-Oct-2019].

[13] R. Mazerik, "Risk Associated with Cookies," *infosecinstitute.com*, Available: resources.infosecinstitute.com/risk-associated-cookies/, November 15 2013. [Online]. [Accessed: 8-Oct-2019].

[14] GitHub Inc., "Securing your webhooks," *developer.github.com*, Available: developer.github.com/webhooks/securing/, 2019. [Online]. [Accessed: 8-Oct-2019].

[15] B. Schneier, "Cryptanalysis of SHA-1," *www.schneier.com*, Available: www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html, February 18 2005. [Online]. [Accessed: 22-Oct-2019].

[16] UBC Launch Pad, "inertia/users.go at v0.5.3," *GitHub*, Available: https://github.com/ubclaunchpad/inertia/blob/v0.5.3/daemon/inertiad/auth/users.go#L150-L210, 2019. [Online]. [Accessed: 27-Nov-2019].

[17] UBC Launch Pad, "inertia/permissions.go at v0.5.3," *GitHub*, Available: https://github.com/ubclaunchpad/inertia/blob/v0.5.3/daemon/inertiad/auth/permissions.go#L81-L82, 2019. [Online]. [Accessed: 27-Nov-2019].

[18] gorilla/websocket, "Potential DoS Vector in gorilla/websocket $\leq$ v1.4.0," *GitHub*, Available: https://github.com/gorilla/websocket/security/advisories/GHSA-jf24-p9p9-4rjh, 2019. [Online]. [Accessed: 28-Nov-2019].

[19] microsoft/go-winio, "Release v0.4.13 go-winio," *GitHub*, Available: https://github.com/microsoft/go-winio/releases/tag/v0.4.13, 2019. [Online]. [Accessed 28-Nov-2019].

[20] dgrijalva/jwt-go, "security: epoch time expiry passes as valid · Issue #277," *GitHub*, Available: https://github.com/dgrijalva/jwt-go/issues/277, 2019. [Online]. [Accessed 28-Nov-2019].

Appendix A: Code of Conduct

We adhered to the following code of conduct throughout our analysis of Inertia. The ethical principles listed are adapted from the 2018 version of "ACM Code of Ethics and Professional Conduct".

A. Contribute to society and to human well-being

Our security analysis of Inertia reduced the probability of the system being exploited by adversaries. We therefore contributed to the well-being of Inertia's end users by ensuring that they can safely build and deploy their software projects.

B. Avoid harm

To avoid harm to Inertia's end users, we performed penetration testing on our own instance such that any disruptions to project builds and deployments affected only us. To ensure that users were not harmed by the complete disclosure of vulnerabilities, we engaged in responsible disclosure so that system owners have time to fix the security issues reported.

C. Be honest and trustworthy

We based our report on honest analysis results and actual findings. In addition, we explicitly indicated the limitations of our conclusions and recommendations.

D. Respect the work required to produce new ideas, inventions, creative works, and computing artifacts

We assigned credit where it is due by providing references to documentations that we consulted for writing our report, or clearly indicated its source.

E. Honor confidentiality

We respected the confidentiality of Inertia's end users by avoiding tampering with client data. All data involved in analysis and penetration testing existed only on our own testing environment. We respected the confidentiality of sensitive project information, such as intermediate vulnerability findings, by communicating amongst ourselves and with the system owners via private chats.

F. Accept and provide appropriate peer review

We worked together to confirm each other's analysis results and gave constructive feedback on the results' validity and significance.

G. Access computing and communication resources only when authorized

We obtained authorization to analyze Inertia by signing a project authorization form with Inertia's system owners. We only analyzed the code developed by Inertia's system owners and not the external libraries used by Inertia (such as Docker and Git APIs in Go).

Appendix B: Responsible Disclosure

A. *Responsible disclosure timeline*

We arranged an in-person meeting with the system owners on Wednesday, December 11th, 2019. We discussed our main findings, and informed the system owner that the final report would be available to the general public 6 months after the final report's due date.

After the arranged meeting, we emailed the system owner a summary of our analysis results with the final report as an attachment.

B. *System owner contact information*

Our primary point of contact is Robert Lin, Inertia's lead developer. He can be reached by email at robert@bobheadxi.dev.