Code Audit Report

Gosling

V 1.1
Amsterdam, November 12th, 2024
Public

Document Properties

Client	Gosling
Title	Code Audit Report
Targets	Gosling changes since 2022 audit Cgosling and FFI boundaries
Version	1.1
Pentester	Morgan Hill
Authors	Morgan Hill, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	October 31st, 2024	Morgan Hill	Initial draft
0.2	October 31st, 2024	Marcus Bointon	Review
0.3	November 7th, 2024	Morgan Hill	Add retesting updates
1.0	November 7th, 2024	Marcus Bointon	1.0
1.1	November 12th, 2024	Morgan Hill	Classify as public

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Radically Open Security B.V. is registered at the trade register of the Dutch chamber of commerce under number 60628081.

Table of Contents

1 Executive Summary

1.1 Introduction

Between August 20, 2024 and November 1, 2024, Radically Open Security B.V. carried out a penetration test for Gosling.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of work

The scope of the penetration test was limited to the following targets:

- Gosling changes since 2022 audit
- Cgosling and FFI boundaries

The scoped services are broken down as follows:

- Code audit cgosling: 2 days
- Code audit gosling changes since 2022 audit: 1 days
- Build script review: 0.5 days
- Set up testing environment: 0.5 days
- Test FFI bindings: 2 days
- Reporting: 1 days
- **Total effort: 7 days**

1.3 Project objectives

ROS will perform an audit of the Gosling libraries with Gosling in order to assess the security of its implementation and FFI bindings. To do so ROS will access Gosling's source code and guide Gosling in attempting to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The security audit took place between August 20, 2024 and November 1, 2024.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 2 High and 1 Low-severity issues.

This audit uncovered two high severity issues, both in the Java bindings. The first is a timing oracle that may be used for de-anonymization CLN-001 (page 10). The second is a denial of service which could lead to de-anonymization CLN-012 (page 12).

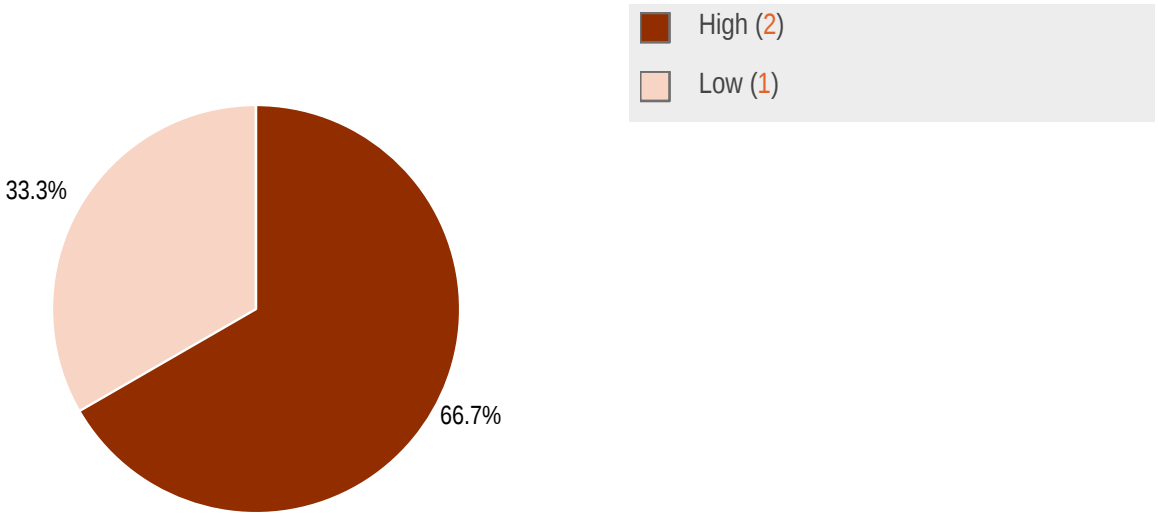
There was one low severity issue. We noted a vulnerable dependency in CLN-003 (page 14); there is no patch available for it at the time of reporting, but Gosling's use of the dependency appears to avoid the weakness.

The core of Gosling is safe Rust and the code quality is generally high.

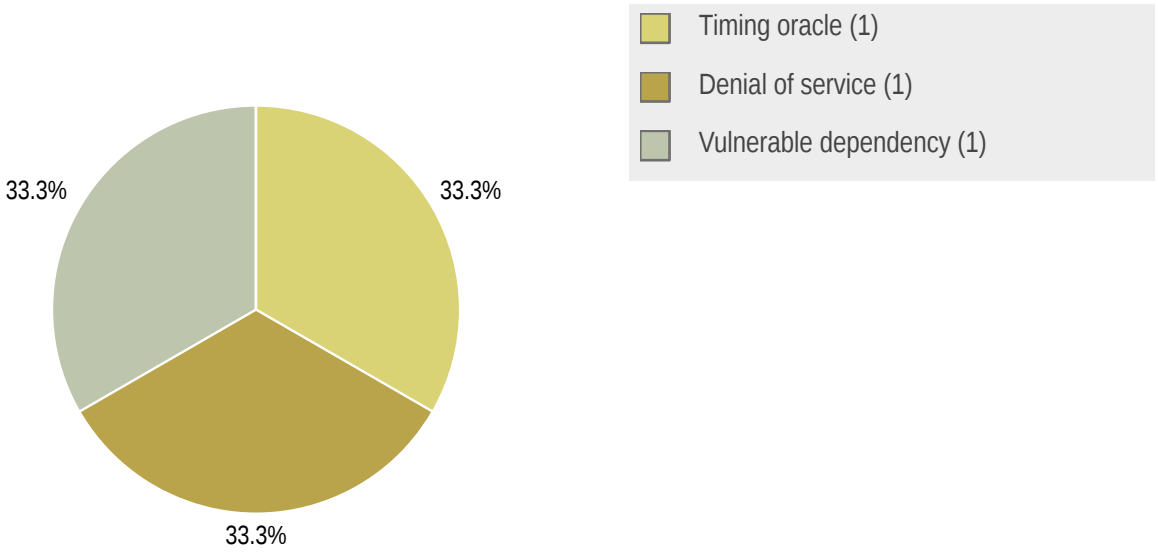
1.6 Summary of Findings

ID	Type	Description	Threat level
CLN-001	Timing oracle	The tcp_pump (used to bridge Java to native sockets) contains a timing oracle, through which an attacker can confirm links between separate identities operating within the same application.	High
CLN-012	Denial of Service	When handling multiple connections via the Java bindings, the Gosling context becomes invalid.	High
CLN-003	Vulnerable dependency	The rsa crate is subject to a timing side channel attack.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
CLN-001	Timing oracle	• Several options have been discussed with the developers, however it is not clear which remedy will strike the correct balance of performance, effectiveness, and avoiding introducing new vulnerabilities. See CLN-001 (page 10) for more details.
CLN-012	Denial of Service	• Access to the PoC code that reproduces this issue has been shared with Gosling's developers. Using this it should be possible to identify the root course, fix it, and develop tests to ensure this denial of service condition does not reoccur.
CLN-003	Vulnerable dependency	• There is currently no patch available. Keep abreast of new developments and update the dependency once a fix is available.

2 Methodology

2.1 Planning

Our general approach during code audits is as follows:

1. **Static Code Analysis**
   We performed static analysis on the Rust code with `cargo geiger` and `cargo audit`. We performed dynamic testing by running existing unit tests with miri, an experimental Rust interpreter that can be used to detect various types of bugs (often in unsafe Rust).
2. **Code Analysis and Fuzzing**
   Manual code review and static code analysis were accompanied by the development of fuzzing targets and developing entirely new targets. We used `cargo fuzz` and designed our targets to conform to the libFuzzer interface, which can be used with a variety of fuzzers including AFL++.
   While reading the code we would occasionally spot interesting code paths and develop small stubs to exercise them. These stubs were then used with stepwise debuggers to understand how the code operates at runtime in greater depth.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.
- **High**
  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**
  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**
  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.

- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Findings

We have identified the following issues:

3.1 CLN-001 — Java bindings tcp_pump timing oracle

Vulnerability ID: CLN-001	**Status:** Resolved
Vulnerability type: Timing oracle	
Threat level: High	

Description:

The `tcp_pump` (used to bridge Java to native sockets) contains a timing oracle, through which an attacker can confirm links between separate identities operating within the same application.

Technical description:

Gosling exposes Java JNI bindings to enable the use of Gosling from Java applications. The JNI interface does not provide a mechanism to create a Java socket from a native operating system socket. To work around this limitation, Gosling's Java bindings proxy between a native socket and a Java socket in a `tcp_pump` thread. The `tcp_pump` thread and its associated state is initiated and shared once within an application.

Shared state is inherently interesting from a security perspective as it often introduces the opportunity for side channels. In this case timing side channels are theoretically present using either lock contention of the mutex or CPU resource contention in the `tcp_pump` thread.

The code for this thread can be found here: https://github.com/blueprint-freespeech/gosling/blob/a82eef8ff54788a18ec5db8b968121b440a0916f/source/bindings/java/GoslingJNI.cpp.handlebars#L183.

The `tcp_pump` thread allocates a dynamic `std:vector` backed buffer. The implementation does not place restrictions on the upper bounds of how much it will read from a socket at a time. The amount of data that can be read from the operating system socket is limited by the socket read buffer, which defaults to around 1MB on modern operating systems. The more data that is read, the longer allocating the buffer and copying into it will take, and the longer the next proxy in the queue must wait before it is serviced by the pump.

By sending large messages to one endpoint, an attacker could block the `tcp_pump` thread for around 40 microseconds. 40 microseconds would be too minuscule on its own to distinguish from noise over the Tor network. However, by flooding a couple of hundred connections, an attacker could block for a couple of milliseconds which is plausibly above the noise floor.

Imagine an application which sends received reports to messages, a target user who operates multiple identities within this application, and an attacker who is connected with the user via two or more of these identities. The attacker suspects that the same person is behind multiple identities and would like to confirm this.

The attacker starts a connection with one identity, which we will call the return channel, and sends a series of messages, recording the round-trip time from message send to receive report received.

Once they have established the latency/jitter baseline of the circuit, the attacker moves to the latency injection phase. The attacker opens many connections to the identity they would like to confirm the link with, flooding it with data. During this phase the attacker sends another series of messages over the return channel and again records the round-trip times.

The attacker then compares the latency injection phase measurements against the baseline measurements. If there is a difference, roughly of the order of the latency the attacker intended to inject, then the identities are likely linked. The attack can be repeated on fresh circuits until the desired level of statistical significance is reached.

Impact:

The application needs to use the Java bindings, support operating multiple concurrent identities, and expose a feature that an attacker can obtain an automatic response from in order to be able to observe the oracle. If these preconditions are met then an attacker may be able to link multiple identities together using timing attacks. These attacks would likely be noisy, increasing the probability of the user noticing. The attack would also require non-trivial resources for the attacker so it is likely to only be used to confirm suspected linked identities.

This oracle is not currently reachable in a practical fashion as it is obscured by CLN-012 (page 12).

Recommendation:

Several options have been discussed with the developers, however it is not clear which remedy will strike the correct balance of performance, effectiveness, and avoiding introducing new vulnerabilities. The PoC code has been made available to the developers to test with here https://git.radicallyopensecurity.com/pcwizz/gosling-java-sidech-poc/-/tree/working?ref_type=heads as the issue is obscured by CLN-012 (page 12) the PoC has not been fully evaluated. The PoC may need more adjustments in order to function once the invalid context issue has been resolved.

Update 2024-11-07 09:00:

Further PoC refinement would be required to make this attack possible. Exploitation of the timing oracle would in fact only be possible if the victim has plenty of bandwidth. One of the issues we observed while testing the PoC was that the effects of bandwidth limitations on a typical domestic internet connection are observed before reaching the point where the latency variations of the oracle become measurable.

The developers have altered the tcp_pump to read only 4k at a time. This significantly increases the number of connections required for the attack to work, making it much less feasible. The code that altered this behaviour can be

seen here: https://github.com/blueprint-freespeech/gosling/blob/e996c0740cdada8e78617fc0a29427e74c19e9af/source/bindings/java/GoslingJNI.cpp.handlebars#L205.

3.2 CLN-012 — Java bindings invalid context

Vulnerability ID: CLN-012	**Status:** Resolved
Vulnerability type: Denial of Service	
Threat level: High	

Description:

When handling multiple connections via the Java bindings, the Gosling context becomes invalid.

Technical description:

While developing a PoC for CLN-001 (page 10), we discovered that the Gosling context becomes invalid when handling multiple connections to an endpoint. The issue occurs when the number of connections is greater than 1, however it does not trigger reliably at 2 connections, and it has been observed to function normally with up to 10 connections. Both contexts in the PoC are invalidated despite connections only being established with one identity.

In the PoC shared with the developers, the error can be observed originating from this line https://git.radicallyopensecurity.com/pcwizz/gosling-java-sidech-poc/-/blob/working/target/Target.java?ref_type=heads#L246.

```
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
alice java.lang.Exception: error: context is invalid
bob java.lang.Exception: error: context is invalid
test@tests-Virtual-Machine gosling-java-sidech-poc %
```

The root cause was not identified during this audit, although possible causes have been discussed with the developers.

Impact:

By making multiple connections to a Java Gosling application, an attacker could trigger a Denial of Service (DoS). In Gosling a DoS is more impactful than normal because it can be used as part of a de-anonymization attack. There are at least two conceivable forms of de-anonymization attack possible using this DoS vulnerability: The first would be to link accounts, similar to CLN-001 (page 10); The second would be to use the DoS to generate a signal visible to traffic analysis by an adversary with privileged position(s) on the network.

Recommendation:

- Access to the PoC code that reproduces this issue has been shared with Gosling's developers. Using this it should be possible to identify the root course, fix it, and develop tests to ensure this denial of service condition does not reoccur.

Update 2024-11-07 09:00:

On discussing this issue with the developer, it became apparent that the bug was in the PoC target. The library must be loaded, and a reference to the library must be maintained, as otherwise the Java garbage collector will drop the library invalidating its uses. The eventual solution was to keep the library as a static member of the target class.

This subtlety should be documented to prevent would-be Gosling App developers from building this bug into their application.

3.3 CLN-003 — Rsa crate dependency timing side channel

Vulnerability ID: CLN-003

Vulnerability type: Vulnerable dependency

Threat level: Low

Description:

The `rsa` crate is subject to a timing side channel attack.

Technical description:

The `rsa` crate is susceptible to a Marvin attack (https://www.redhat.com/en/blog/marvin-attack) tracked here https://rustsec.org/advisories/RUSTSEC-2023-0071.html, which takes advantage of non-constant time operations in `RSAES-PKCS1-v1_5`. The Gosling crates do not directly use the vulnerable methods.

Impact:

The jitter of the Tor network makes such a vulnerability even harder to exploit, but if it were to be exploited then confidentiality could be compromised.

Recommendation:

- There is currently no patch available. Keep abreast of new developments and update the dependency once a fix is available.

4 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

4.1 NF-002 — Dependency management

Gosling appears to stay up-to-date with its dependencies. The `Cargo.lock` is committed to the repository as is considered good practice in the Rust ecosystem (https://doc.rust-lang.org/cargo/faq.html#why-have-cargolock-in-version-control). Attention should be paid to ensuring that `cargo update` is run regularly to keep the `Cargo.lock` fresh and ensure that development and testing is performed with the latest dependencies that meet the constraints in the `Cargo.toml`. Tooling such as Dependabot or RenovateBot may be helpful to catch any bugs reported in dependencies.

4.2 NF-007 — semgrep did not produce any results

semgrep is a popular static analysis tool that scans for common weakness in code. Running against Gosling it did not raise any warnings.

4.3 NF-008 — Pointers from C guarded with ensure_not_null macro

Gosling ensures that pointers from the FFI boundary are not `null` using the `ensure_not_null` macro before attempting to operate on them. This prevents `null` pointer dereferences from occurring. There is still a responsibility on the caller to ensure that the pointer is valid for the intended type and lifetime. For strings, Gosling additionally validates that the length is not 0.

4.4 NF-009 — Dynamic library threat model

When not being used from Rust, Gosling is consumed as a dynamic library (`dylib`, `so`, or `dll` on macOS, Linux, and Windows respectively). For the Java bindings there is an additional layer of redirection to bridge from the JNI to cgosling. The nature of dynamic libraries is that control over the library loading path allows injecting code between the application and its libraries. In Gosling's case it would be possible to alter, view, drop, or create messages on the local system using the user's identity.

It is reasonable to discount this threat for most users, as they should be able to trust the environment in which they are running a Gosling application. A similar threat would also apply to the Tor binary that Gosling invokes (when not built to use Arti). Users must be aware of provenance of the libraries and executables on their system, and any other users they may share the system with, as these are inherently trusted. It may be advisable for a user who must use a multi-user system to install Tor, Gosling, and the application into their home/user directory and execute the application with a restricted path.

The Gosling project does not distribute binaries itself, so is not in a position to sign binaries. On macOS and windows signing library objects is a supported and common practice. On Linux there is less support for signing libraries/binaries, although the package the objects were installed from was likely signed if it came a via a package manager; libraries are Linux are verified at installation time but not on load. Developers building applications based on Gosling should consider distributing signed versions of Gosling and Tor with their applications.

5 Future Work

- **semver-checks**

  `cargo-semver-checks` can be a very useful tool for ensuring that the public interfaces of crates remain backwards compatible.
- **Clippy**

  Clippy is a sophisticated linting tool for Rust that can highlight some forms of unsoundness. Clippy does not currently spot any soundness issues in Gosling, however it does emit some warnings. Achieving a warning-free Clippy run may be desirable to remove noise from the output i.e. everything it outputs is new and interesting.
- **Retest of findings**

  When mitigations for the vulnerabilities described in this report have been deployed, a repeat test should be performed to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**

  Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.

6 Conclusion

We discovered 2 High and 1 Low-severity issues during this penetration test.

Gosling is a protocol and library for building peer-to-peer applications over the Tor network. This audit focused on Gosling's implementation and bindings from its native Rust into other languages such as Python and Java. The work has involved manual code review, threat modeling, and dynamic experimentation.

The quality of the Rust code base is very high; none of the findings reported here relate to the Rust code. The core `gosling` library uses safe Rust exclusively, and is already equipped with fuzz testing. Unsafe Rust is used only at the boundaries, for example in `cgosling` which implements the intermediate bindings to C. Via a series of templates, `cgosling` is mapped into more developer-friendly bindings for Java, CPP, Python, and C.

During this audit, the most interesting of these bindings has been for Java. The Java bindings are created with the Java Native Interface implemented in CPP. It is in this JNI layer that the two high-severity findings of this report were found. The first became apparent after intensive reading and modeling for the code that translates Java sockets to native sockets. The second was discovered while attempting to implement a proof of concept for the first.

Before work could begin on testing the bindings, both the Java and Python examples had to be fixed as they no longer ran. These bindings had not received the same level of attention in terms of testing as the others. It is a key indication of this audit that more effort needs to be invested in testing and maintaining the Java and Python bindings if they are to be widely used.

Communication with the developers has been essential to gather context and test assumptions throughout the process. For their part the developers have been responsive and insightful. They have demonstrated a clear interest in security, this being their second audit of Gosling with ROS. We look forward to opportunities to work with them again in the future.

This work was made possible with funding from NLnet under the NGIR program. We appreciate this opportunity to contribute to the security of privacy-respecting technologies.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process – this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing team

Morgan Hill	Morgan is a seasoned security consultant with a background in IoT and DevOps. He currently specialises in Rust and AVoIP.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.

Front page image by dougwoods (https://www.flickr.com/photos/deerwooduk/682390157/), "Cat on laptop", Image styling by Patricia Piolon, https://creativecommons.org/licenses/by-sa/2.0/legalcode.