KUDELSKI SECURITY

Security Assessment for Swim Pool

Findings and Recommendations Report Presented to:

Swim Protocol

December 13, 2021

Version: 1.0.1- Final for Public Release

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

FINAL – FOR PUBLIC RELEASE

TABLE OF CONTENTS

TABLE OF CONTENTS 2
LIST OF FIGURES 3
LIST OF TABLES 3
EXECUTIVE SUMMARY 4
- Overview 4
- Key Findings 4
- Scope and Rules of Engagement 5

TECHNICAL ANALYSIS & FINDINGS 6
- Findings 7
- Technical analysis 7
- Technical Findings 8
  - General Observations 8
  - Decimal, Panics and Unwraps 9
  - Order of operations lose precision 11
  - Mint authority of token mint accounts is not checked 13
  - Initial depth of unknown balance inconsistent with documentation 14
  - Documentation shows depth estimation diverging for relative error used 16
  - Undocumented branches in `calculate_unknown_balance` 17
  - Undocumented fee calculation 18

METHODOLOGY 19
- Kickoff 19
- Ramp-up 19
- Review 20
- Code Safety 20
- Technical Specification Matching 20
- Reporting 20
- Verify 21
- Additional Note 21
- The Classification of identified problems and vulnerabilities 21
  - Critical – a vulnerability that will lead to loss of protected assets 22
  - High - A vulnerability that can lead to loss of protected assets 22
  - Medium - a vulnerability that hampers the uptime of the system or can lead to other problems 22
  - Low - Problems that have a security impact but does not directly impact the protected assets 22
  - Informational 22
- Tools 23

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

RustSec.org 23
KUDELSKI SECURITY CONTACTS 24

LIST OF FIGURES

Figure 1: Findings by Severity 6
Figure 2: Methodology Flow 19

LIST OF TABLES

Table 1: Scope 5
Table 2: Findings Overview 7

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

EXECUTIVE SUMMARY

Overview

Swim Protocol engaged Kudelski Security to perform a Security Assessment for Swim Pool.

The assessment was conducted remotely by the Kudelski Security Team. Testing took place on November 01 - November 12, 2021, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarises the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following are the major themes and issues identified during the testing period. Within the findings section, these, along with other items, should be prioritised for remediation to reduce the risk they pose.

- KS-SWIMPOOL-01 – Decimal, Panics and Unwraps
- KS-SWIMPOOL-02 – Order of operations lose precision
- KS-SWIMPOOL-03 – Mint authority of token mint accounts is not checked
- KS-SWIMPOOL-04 – Initial depth of unknown balance inconsistent with documentation
- KS-SWIMPOOL-05 – Documentation shows depth estimation diverging for relative error used
- KS-SWIMPOOL-06 – Undocumented branches in `calculate_unknown_balance`
- KS-SWIMPOOL-07 – Undocumented fee calculation

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was supportive and open to discussing the design choices made

Based on the source code, the validity of the code was verified and confirmed that the intended functionality was implemented correctly and to the extent that the state of the repository allowed. As of the issuance of this report, all findings are resolved to our expectations.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Scope and Rules of Engagement

Kudelski performed a Security Assessment for Swim Pool. The following table documents the targets in scope for the engagement. No other systems or resources were in scope for this assessment.

The source code was supplied through a private repository at https://gitlab.com/btblock-cybersec/swim/pool with the commit hash 6e73232b0a7e6934103c50cb6ef830fb2f91de9b. The code was then reaudited at commit hash 7e5a214efc630e7f158fc9ca711a4dd6666a92a2.

Files included in the code review
src ├── amp_factor.rs* ├── common.rs* ├── decimal.rs* ├── entrypoint.rs* ├── error.rs* ├── instruction.rs* ├── invariant.rs* ├── lib.rs* ├── pool_fee.rs* ├── processor.rs* └── state.rs*

Table 1: Scope

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

TECHNICAL ANALYSIS & FINDINGS

During the Security Assessment for Swim Pool, we discovered:

- 7 findings with an INFORMATIONAL severity rating.

The following chart displays the findings by severity.

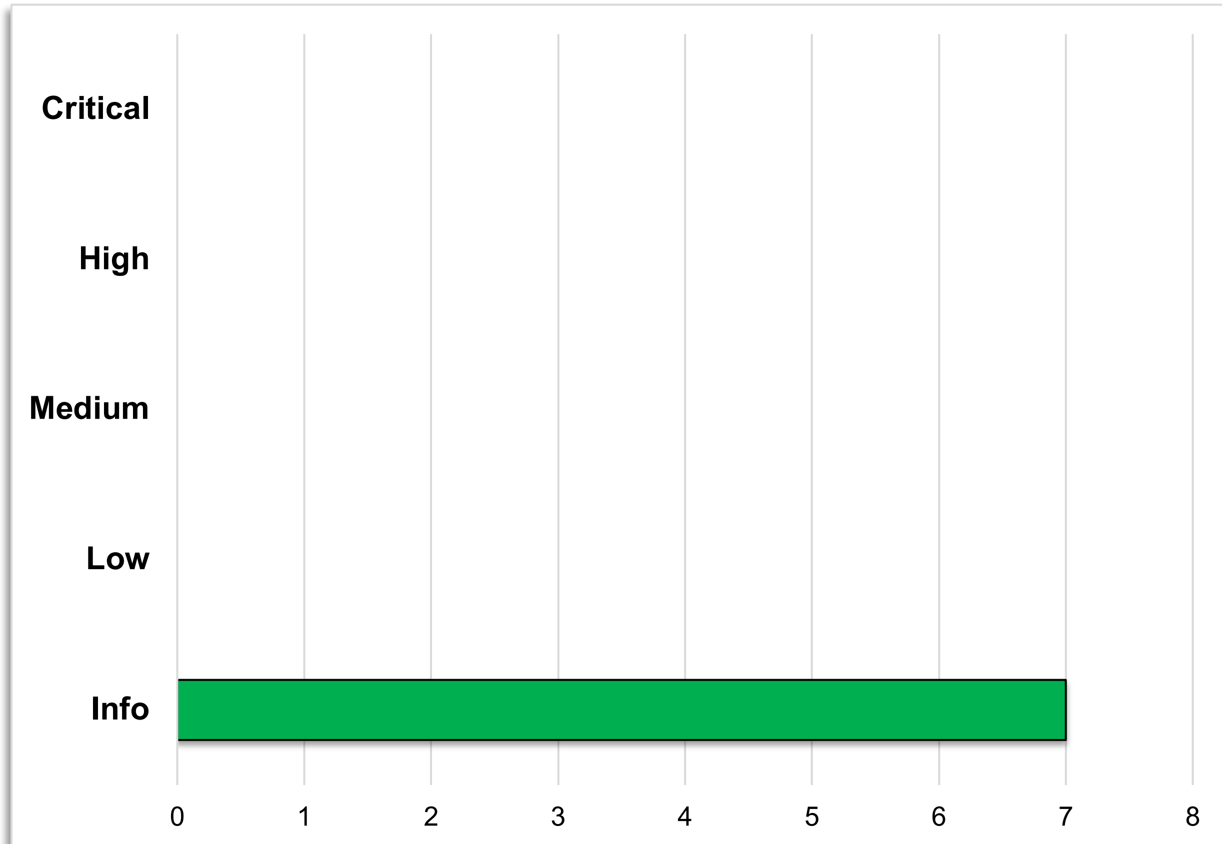


Figure 1: Findings by Severity

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Findings

The *Findings* section provides detailed information on each finding, including discovery methods, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-SWIMPOOL-01	**Informational**	Decimal, Panics and Unwraps
KS-SWIMPOOL-02	**Informational**	Order of operations lose precision
KS-SWIMPOOL-03	**Informational**	Mint authority of token mint accounts is not checked
KS-SWIMPOOL-04	**Informational**	Initial depth of unknown balance inconsistent with documentation
KS-SWIMPOOL-05	**Informational**	Documentation shows depth estimation diverging for relative error used
KS-SWIMPOOL-06	**Informational**	Undocumented branches in `calculate_unknown_balance`
KS-SWIMPOOL-07	**Informational**	Undocumented fee calculation

Table 2: Findings Overview

Technical analysis

Based on the source code, the validity of the code was verified and confirmed that the intended functionality was implemented correctly and to the extent that the state of the repository allowed. Many further investigations were made, which concluded that they did not pose a risk to the application. They were:

- No potential authorisation issues were observed
- No internal unintentional unsafe references
- No large memory allocations
- No unjustified drop implementations

Based on formal verification, we conclude that the code implements the documented functionality to the extent of the code reviewed.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Technical Findings

General Observations

Swim's pool program is an Automated Market Maker aiming to enable the exchange of a fixed number of stable coins in the form of tokens in the Solana blockchain. The program manipulates accounts, each holding amounts of a distinct token, in addition to a liquidity pool token. The latter is associated with the pool's depth, fees and rewards, and can be exchanged for other tokens of the pool as well. Token prices are calculated and set dynamically according to StableSwap.

Code Quality

All necessary precautions are taken to ensure the accounts' integrity and correct authorisation regarding security. The pool's state account data are correctly checked and de-serialised. Instructions are separated between initialisation, governance and finance, with each section being coherent and easy to follow.

The implementation of price calculations takes place in a separate file. Although the separation is beneficial, the invariant code lacks documentation and clarity, making it harder to read and navigate. The invariant's functions were checked against the documentation provided, with some inconsistencies listed as found below.

It should also be noted that commented and //TODO code was also included in the version of the code reviewed.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Decimal, Panics and Unwraps

Finding ID: KS-SWIMPOOL-01
Severity: **Informational**
Status: **Rejected**

Description

The project chose to provide its implementation of decimal representation. Arithmetic functions (either derived or explicit) involve custom types and their uses, resulting in many warnings while linting. Although arithmetic functions are intended and designed to keep numbers within bounds, unchecked arithmetic should be avoided. We understand that `DecimalU64` is used for saving space as opposed to Rust's `Decimal`. However, testing seems short. In addition, the code includes unchecked unwraps and explicit panics.

Proof of issue

```
$ cargo clippy -- -A clippy::all -W clippy::integer_arithmetic -W
clippy::integer_division
...
warning: 404 warnings emitted
```

File name: src/invariant.rs **Line number:** 35

```
fn fast_round(decimal: Decimal) -> AmountT {
    //TODO no rounding to preserve compute budget for now

    // const ONE_HALF: Decimal = Decimal::from_parts(5, 0, 0, false, 1);
    // AmountT::from((decimal + ONE_HALF).trunc().to_u128().unwrap())

    //due to rounding errors we can get negative values here, hence the
unwrap_or
    //decimal.to_u128().unwrap_or(0).into()
    decimal.to_u128().unwrap().into()
}
```

File name: src/decimal.rs

```
// Line 506
            fn add(self, other: Self) -> Self::Output {
                self.checked_add(other)
                    .unwrap_or_else(|| panic!("Overflow while adding {:?}
{:?}", self, other))
            }
// Line 561
            fn sub(self, other: Self) -> Self::Output {
                self.checked_sub(other)
                    .unwrap_or_else(|| panic!("Underflow while subtracting
{:?} {:?}", self, other))
            }
// Line 606
            fn mul(self, other: Self) -> Self::Output {
                self.checked_mul(other)
```

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

```
                .unwrap_or_else(|| panic!("Overflow while multiplying
{:?} {:?}", self, other))
            }
// Line 659
            fn div(self, other: Self) -> Self::Output {
                self.checked_div(other)
                    .unwrap_or_else(|| panic!("Division by zero while
dividing {:?} {:?}", self, other))
            }
```

Severity and Impact summary

Commented code removes from the code's otherwise good clarity. Unchecked unwraps and panics interrupt the normal and expected execution of transactions. Finally, relying on external crates (such as `uint` for `U128`) adds to uncertainty, complexity, instability, and size of the binary.

Recommendation

Since invariant functions mostly use Rust's native `Decimal` type, it is recommended to use native types throughout. To save space, the structures could be kept without the additional functionality. `decimal.rs` also includes types that are not used in the project, like DecimalU128, whose removal, macros could be avoided, simplifying the code. In general, unused code should be avoided, reducing the final binary's size.

Panics should be avoided using checks, `Option` and/or `Result`.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Order of operations lose precision

Finding ID: KS-SWIMPOOL-02
Severity: **Informational**
Status: **Open**

Description

When multiple decimal arithmetic functions are combined in the current implementation, precedence is sometimes given to the division using parentheses. This loses precision that can be preserved by changing the operations' order.

A quick example to demonstrate this is that:

```
Decimal::TEN * (Decimal(1,28)) / Decimal::TEN) == Decimal::ZERO &&
(Decimal::TEN * Decimal(1,28)) / Decimal::TEN) == Decimal:: Decimal(1,28)
```

Proof of issue:

File name: src/invariant.rs

```
// Line 369
            Decimal::from(initial_depth.to_u128().unwrap())
                * (Decimal::from(sum_updated_balances.as_u128()) /
Decimal::from(sum_pool_balances.as_u128())),
// Line 375
                        Decimal::from(balance.as_u128())
                            * (Decimal::from(sum_updated_balances.as_u128())
                                /
Decimal::from(sum_pool_balances.as_u128())),
// Line 426
            let lp_amount =
fast_round(Decimal::from(lp_total_supply.as_u128()) * (user_depth /
initial_depth));
            let governance_depth = total_fee_depth * (governance_fee /
total_fee);
// Line 438
                dec_sub_given_order(is_add, updated_depth, initial_depth) /
initial_depth
                    * Decimal::from(lp_total_supply.as_u128()),
// Line 458
        let updated_depth = initial_depth
            * (Decimal::from((lp_total_supply - burn_amount).as_u128()) /
Decimal::from(lp_total_supply.as_u128()));
// Line 477
            let governance_depth = total_fee_depth * (governance_fee /
total_fee);
            let governance_mint_amount = fast_round(
                governance_depth
                    * (Decimal::from((lp_total_supply -
burn_amount).as_u128()) / (updated_depth - governance_depth)),
            );
// Line 510
```

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

```
            let reciprocal_decay = pool_balances_times_n
                .iter()
                .fold(Decimal::one(), |acc, &pool_balance_times_n| {
                    acc * (depth / pool_balance_times_n)
                });
// Line 537
        let reciprocal_decay = known_balances.iter().fold(Decimal::one(),
|acc, &known_balance| {
            acc * (depth / Decimal::from((known_balance * n).as_u128()))
        });
```

File name: src/amp_factor.rs **Line number:** 72

```
            let delta = value_diff * (time_since_initial /
total_adjustment_time);
```

Severity and Impact summary

Operation order can reduce precision.

Recommendation

Values should be inflated first before being divided to maximise precision.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Mint authority of token mint accounts is not checked

Finding ID: KS-SWIMPOOL-03
Severity: **Informational**
Status: **Remediated**

Description

The liquidity pool's mint account is checked during initialisation for the correct `mint_authority` and an empty `freeze_authority`. These are not checked for the token mint accounts.

Proof of issue

File name: src/processor.rs **Line number:** 138

```
        let token_decimals: [_; TOKEN_COUNT] = create_result_array(|i| ->
Result<_, ProgramError> {
            let mint_decimals =
Self::check_program_owner_and_unpack::<MintState>(token_mint_accounts[i])?.de
cimals;
            // ... no check is performed in the following code
```

Severity and Impact summary

Minters of token accounts could affect the total supply of tokens by minting or burning without updating the pool.

Recommendation

Unless the token accounts are somehow trusted, check for the existence of a `freeze_authority` in `token_mint_accounts`. Another check could be for the uniqueness of `mint_authority` of `token_mint_accounts` to avoid the same mint for two accounts.

Remediation

In practice, the pools will correspond to trusted stable coin token accounts.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Initial depth of unknown balance inconsistent with documentation

Finding ID: KS-SWIMPOOL-04
Severity: **Informational**
Status: **Remediated**

Description

According to the documentation provided:

Even though Dn seems like the canonical choice for the initial guess of xj, we have to use D. To see why to look at $G(xj)$ and consider a pool that's arbitrarily far from equilibrium (i.e. $\forall i \neq j : xi < \epsilon$). The numerator of $G(xj)$ would grow arbitrarily large, while the denominator would tend towards $xj + D/A - D$ and would hence be 0 or even harmful if $xj \leq D - D/A$. Since A is an arbitrary parameter, D/A can be arbitrarily small. Therefore we have to choose $xj = D$ to ensure a practical value at all times.

This is not consistent with the actual implementation.

Proof of issue

Filename: src/invariant.rs

```
// Line 303
        let unknown_balance = Self::calculate_unknown_balance(
            &known_balances,
            initial_depth,
            amp_factor,
            if is_exact_input {
                pool_balances[index]
            } else {
                AmountT::zero()
            },
        )?;
// Line 462
        let unknown_balance =
            Self::calculate_unknown_balance(&known_balances, updated_depth,
amp_factor, pool_balances[output_index])?;
```

Severity and Impact summary

The inconsistent initial value during unknown value calculation could result in calculations other than expected.

Recommendation

Use the pool's depth as an initial value for unknown value estimation, or update the documentation accordingly.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Remediation

Updates to the initial values provided were made, and clarification was provided for the differences between documentation, in which calculations are agnostic to a pool's state and in practice scenarios where the pool's previous depth can be used as initial depth.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Documentation shows depth estimation diverging for relative error used

Finding ID: KS-SWIMPOOL-05
Severity: **Informational**
Status: **Open**

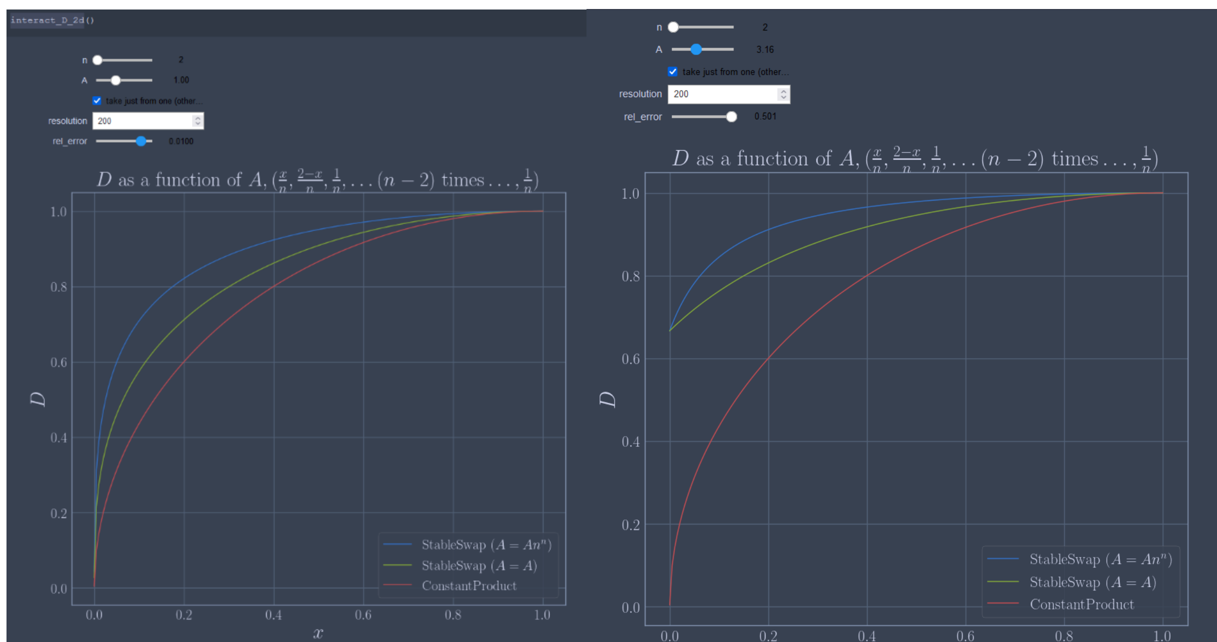
Description

For a relative error of 0.5, used currently, depth estimation seems to diverge for small values of *x*.

Proof of issue

File name: src/invariant.rs **Line number:** 507

```
        while abs_difference(depth, previous_depth) > Decimal::new(5, 1) {
```



Severity and Impact summary

If similar to the documentation, depth estimation diverges from the expected value.

Recommendation

Use a smaller value for the relative error at the cost of a few more iterations.

Note

This issue was initially listed as Medium but moved to Informational after the development team pointed out that depth is calculated to the pool's closest token subdivision in practice.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Undocumented branches in `calculate_unknown_balance`

Finding ID: KS-SWIMPOOL-06
Severity: **Informational**
Status: **Remediated**

Description

A few lines of code seem to branch out into some calculations whose clarity is lacking:

Proof of issue

Filename: src/invariant.rs **Line number:** 545

```
        let numerator_fixed = if reciprocal_decay < Decimal::from(u32::MAX) {
            (U192::from(
                (reciprocal_decay * (depth / Decimal::from(TOKEN_COUNT)))
                    .to_u128()
                    .unwrap(),
            ) * U192::from((depth / amp_factor *
Decimal::from(u32::MAX)).to_u128().unwrap()))
                / U192::from(u32::MAX)
        } else {
            (((U192::from(reciprocal_decay.to_u128().unwrap())
                * U192::from((depth /
Decimal::from(TOKEN_COUNT)).to_u128().unwrap()))
                * U192::from(depth.to_u128().unwrap()))
                / U192::from((amp_factor *
Decimal::from(u32::MAX)).to_u128().unwrap()))
                / U192::from(u32::MAX)
        };
```

Testing the code separately, it seems the two branches yield different outputs depending on the condition of `if reciprocal_decay < Decimal::from(u32::MAX)`. The connection between `u32::MAX` and the pool's operations is not clear.

Severity and Impact summary

Lacking clarity in `calculate_unknown_balance`.

Recommendation

Please document why branching and re-scaling take place.

Remediation

The code was commented on in a later version of the code. Both branches multiply reciprocal_decay * depth / token_count. The former branch represents the case where multiplication can be safely performed, while the latter inflates the numerator to preserve precision.

Further isolated testing of this code is recommended.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Undocumented fee calculation

Finding ID: KS-SWIMPOOL-07
Severity: **Informational**
Status: **Open**

Description

With some documentation, it would be nice to accompany the fee calculations, which seem to be the most complicated part of the invariant calculations.

Severity and Impact summary

Fee calculations affect the invariant in non-trivial ways.

Recommendation

Extend Jupyter notebook to include how the fees are expected to be deducted.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

METHODOLOGY

Kudelski Security uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

Figure 2: Methodology Flow

Kickoff

The project is kicked all of the sales processes has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project and the responsibilities of participants. During this meeting, we verified the scope of the engagement and discussed the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff, there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artefacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on a particular project. This can include the steps needed for familiarity with the codebase or technological innovation utilised. This may include, but is not limited to:

- Reviewing previous work in the area, including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyse the project for flaws and issues that impact the security posture. Depending on the project, this may include an architecture analysis, a code review, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance with the code with the provided technical documentation

The review for this project was performed using manual methods and utilising the reviewer's experience. No dynamic testing was performed. Only custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analysed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behaviour
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general and not comprehensive, meant only to understand the issues we are looking for.

Technical Specification Matching

We analysed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

Kudelski Security delivers a PDF draft report containing an executive summary, technical details, and observations about the project.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

The executive summary contains an overview of the engagement, including the number of findings and a statement about our general risk assessment of the project as a whole. We may conclude that the overall risk is low, but depending on what was assessed, we may conclude that more scrutiny of the project is needed.

We not only report security issues identified but also informational findings for improvement categorised into several buckets:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related but are general best practices and steps that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can create a public report that can be shared and distributed to a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or the delivery of the draft report. We will verify any fixes within a window of time specified in the project. After the fixes have been verified, we will change the finding status in the report from open to remediate.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits and the agreement's scope.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination.

The Classification of identified problems and vulnerabilities

There are four severity levels of an identified security vulnerability.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Critical – a vulnerability that will lead to loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - A vulnerability that can lead to loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that can not be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Tx signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - a vulnerability that hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard, or non-peer-revied crypto functions
- Program crashes leaves core dumps or writes sensitive data to log files

Low - Problems that have a security impact but does not directly impact the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

Tools used during the code review and assessment

- Rust – cargo tools
- IDE modules for Rust and analysis of source code
- Cargo audit, which uses https://rustsec.org/advisories/ to find vulnerabilities cargo.

RustSec.org

About RustSec

The RustSec Advisory Database is a repository of security advisories filed against Rust crates published and maintained by the Rust Secure Code Working Group.

The RustSec Tool-set used in projects and CI/CD pipelines

'cargo-audit' - audit Cargo.lock files for crates with security vulnerabilities.

'cargo-deny' - audit `Cargo.lock` files for crates with security vulnerabilities, limit the usage of particular dependencies, their licenses, sources to download from, detect multiple versions of same packages in the dependency tree and more.

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION

© 2021 Kudelski Security, Inc. Confidential and Proprietary. All Rights Reserved.