Raydium Security Code Review Assessment

Raydium Project / Solana Deployment

11 May 2021
Version: 1.0

Presented by:
Kudelski Security Research Team
Kudelski Security – Nagravision SA

Corporate Headquarters
Kudelski Security – Nagravision SA
Route de Genève, 22-24
1033 Cheseaux sur Lausanne
Switzerland

DOCUMENT PROPERTIES

Version:	1.0
File Name:	Raydium_Research_Report_v1.0
Publication Date:	11 May 2021
Confidentiality Level:	Public
Document Owner:	Scott Carlson
Document Recipient:	Raydium Project
Document Status:	Approved

TABLE OF CONTENTS

EXECUTIVE SUMMARY 6
1.1 Engagement Limitations 6
1.2 Engagement Analysis 6
1.3 Observations 7
1.4 Issue Summary List 8
2. METHODOLOGY 9
2.1 Kickoff 9
2.2 Ramp-up 9
2.3 Review 9
2.4 Reporting 10
2.5 Verify 11
2.6 Additional Note 11
3. GENERAL OBSERVATIONS 12
3.1 Long files and functions 12
3.2 Error handling 12
3.3 Copy-paste code 13
3.4 Code cleaning 15
3.5 Dead code 16
4. TECHNICAL DETAILS 16
4.1 Compiler warnings 16
4.2 Failed test 18
4.3 Low test coverage 20
4.4 Code formatting 22
4.5 Bad code practice 23
4.6 Vulnerabilities from RustSec Advisory Database 25
4.7 Zero out sensitive data 27
4.8 Conditional compilation for little endian only 29
4.9 Invalid reference type 31
4.10 Fibonacci integer overflow 32
4.11 Unchecked power functions 34
4.12 Unchecked conversion from internal pc lot size to srm pc lot size 35

APPENDIX A: ABOUT KUDELSKI SECURITY 38
APPENDIX B: DOCUMENT HISTORY 39
APPENDIX C: SEVERITY RATING DEFINITIONS 40

TABLE OF FIGURES

Figure 1 Issue Severity Distribution 7
Figure 2 Methodology Flow 9

EXECUTIVE SUMMARY

Kudelski Security ("Kudelski"), the cybersecurity division of the Kudelski Group, was engaged by the Raydium Project to conduct an external security assessment in the form of a security code review of the Raydium AMM and Staking Smart Contracts application on the Solana blockchain.

The assessment was conducted remotely by the Kudelski Security Team from March 15, 2021, to April 9, 2021, and focused on the following objectives:

1. To help the Client to better understand its security posture on the external perimeter and identify risks in its infrastructure.
2. To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
3. To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the tests performed and findings in terms of strengths and weaknesses. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to exploit each vulnerability, and recommendations for remediation.

1.1 Engagement Limitations

The architecture and code review are based on the documentation and code provided by Raydium. The code resides in a two private repositories at https://github.com/raydium-io/raydium-staking and https://github.com/raydium-io/raydium-amm (in which only the serum-amm directory was in-scope)

The reviews are based on the commit hashes:

Raydium-staking: 6bfdfba139abc0e7457356ec510ef57f7cc24e14

Raydium-amm: 300c232dd362ede00667df8193bdcab677cf23f3

All third-party libraries were deemed out-of-scope for this review and are expected to work as designed. We have when needed based on the criticality of the dependency looked at the current state of the crate included.

1.2 Engagement Analysis

This engagement was comprised of a code review including reviewing how the architecture has been implemented as well as any security issues. The architecture implementation review was based on the documentation and the information retrieved through communication between the Raydium team and the Kudelski Security team. The implementation review concluded that the application implementation is well thought out, designed well, and uses the Solana features as designed.

The code review was conducted by the Kudelski Security team on the code provided by Raydium, in the form of a Github repository. The code review focused on the handling of secure and private information handling in the code.

As a result of our work, we identified **0 High**, **4 Medium**, **5 Low**, and **3 Informational** findings.

The findings revolve around memory handling in systems handling secure information and overflow issues

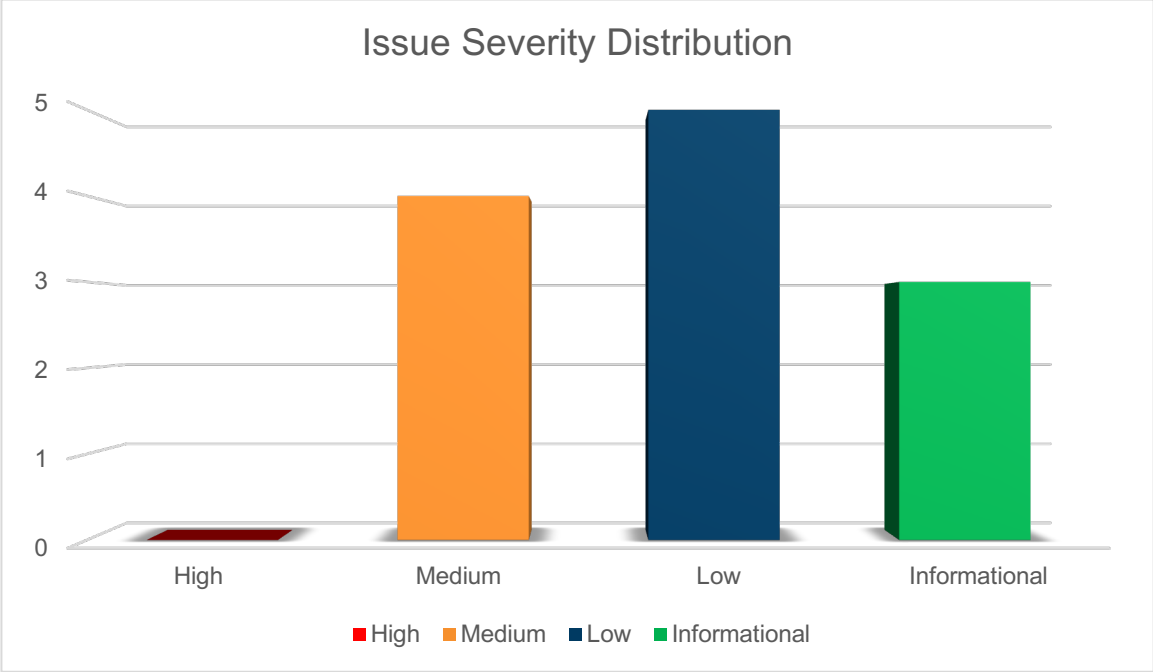

Figure 1 Issue Severity Distribution

1.3 Observations

The Raydium code for both Staking and AMM are quickly evolving. Based on this we could see that it was in a varying state of maturity. We suggest that some refactoring is made to remove the challenges from large files and functions, a more consistent error handling model and that duplicated code is refactored in such a way that the future maintenance of the code base will be less error prone.

We could not find any obvious memory leaks nor any unsafe concurrency conditions. This is a good sign of code maturity.

Some design choices are suboptimal but functional based on the environment and limitations on the application. One such example is the handling of critical errors that results in an execution panic. This results in that the failed transaction will revert to its original state. The flipside is that if no measures for clearing out sensitive data is made during the exit, this will be remaining in computer memory open for a memory scanning attack.

During our initial review, it was noted by the review team that we assumed that the account verification and ownership would be handled outside of the Raydium infrastructure, but after further discussion with the core team, this assumption was changed, and we agree with the recommendation that ownership, permission, and derived addresses always be verified and checked prior to any execution of transaction. This applies to any project building on Solana.

1.4 Issue Summary List

ID	SEVERITY	FINDING
KS-RAYDIUM-F-01	Informational	Compiler warnings
KS-RAYDIUM-F-02	Low	Failed test
KS-RAYDIUM-F-03	Low	Low test coverage
KS-RAYDIUM-F-04	Informational	Code formatting
KS-RAYDIUM-F-05	Informational	Bad code practice
KS-RAYDIUM-F-06	Low	Vulnerabilities from RustSec Advisory Database
KS-RAYDIUM-F-07	Medium	Zero out sensitive data
KS-RAYDIUM-F-08	Low	Conditional compilation for little endian only
KS-RAYDIUM-F-09	Low	Invalid reference type
KS-RAYDIUM-F-10	Medium	Fibonacci integer overflow
KS-RAYDIUM-F-11	Medium	Unchecked power functions
KS-RAYDIUM-F-12	Medium	Unchecked conversion from internal pc lot size to srm pc lot size

2. METHODOLOGY

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

Figure 2 Methodology Flow

2.1 Kickoff

The project is kicked off when the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It is 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 artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

2.2 Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. 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

2.3 Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, 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
2. Review of the code written for the project

3. Compliance of the code with the provided technical documentation.

The review for this project was performed using manual methods and tools, utilizing the experience of the reviewer. No dynamic testing was performed, only the use of 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 analyzed the provided code, checking for issues related to the following categories:

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

This list is general list and not comprehensive, meant only to give an understanding of the issues we are looking for.

Cryptography

We analyzed the cryptographic primitives and components as well as their implementation. We checked in particular:

- Matching of the proper cryptographic primitives to the desired cryptographic functionality needed
- Security level of cryptographic primitives and their respective parameters (key lengths, etc.)
- Safety of the randomness generation in general as well as in the case of failure
- Safety of key management
- Assessment of proper security definitions and compliance to use cases
- Checking for known vulnerabilities in the primitives used

Technical Specification Matching

We analyzed 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

2.4 Reporting

Kudelski Security delivers a preliminary report in PDF format that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as 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 categorized into several buckets:

- 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 agree on the creation of a public report that can be shared and distributed with a larger audience.

2.5 Verify

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

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

2.6 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 along with the scope of the agreement.

While assessment the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. These is a solid baseline for severity determination. Information about the severity ratings can be found in **Appendix C** of this document.

3. GENERAL OBSERVATIONS

3.1 Long files and functions

Description

Code files are very large, and functions are very long.

Proof of Issue

Top 3 of largest files:

1. processor.rs in serum-amm/amm contains 3232 lines.
2. lib.rs in serum-amm/crank contains 3187 lines.
3. main.rs in raydium-staking/crank contains 1200 lines.

Recommendation

- Refactor large functions by extracting code into separate functions.
- Move code to different files.

References

- The Rust Programming Language – Separating Modules into Different Files: https://doc.rust-lang.org/book/ch07-05-separating-modules-into-different-files.html
- The Rust Programming Language – Refactoring to Improve Modularity and Error Handling: https://doc.rust-lang.org/book/ch12-03-improving-error-handling-and-modularity.html#separation-of-concerns-for-binary-projects
- Refactoring Guru – Long Method: https://refactoring.guru/smells/long-method

3.2 Error handling

Description

Returning Result and Option objects from a function that may cause errors is a very good practice. However, panic should only occur due to internal errors such as I/O errors. Usage errors caused by invalid input from users or external services should be propagated to the main error handler instead of causing panic.

Panic from Result and Option objects can occur from unchecked calls to unwrap() and expect().

Proof of Issue

Filename: serum-amm/amm/src/math.rs

Beginning Line Number: 187

```
pub fn convert_in_price(val: u64, pc_lot_size: u64) -> u64 {
    let price = val.checked_mul(pc_lot_size).unwrap();
    price
}
```

The use of checked_mul() to avoid overflow is very good practice. But the unwrap causes an instant panic. Instead, convert_in_price() should just return the Option from checked_mul() directly or return a Result object with an error when an integer overflow occurs.

Recommendation

Propagate errors to avoid panicking while unwrapping as follows:

- For Result objects the ? operator can be used to unwrap the value of the result. If the result contains and error it will immediately be returned by the calling function.

  Example:

  ```
  let checked_value = get_some_result()?;
  ```

- For Option objects can either be transformed into Result object using ok_or() or ok_or_else() or handled immediately using unwrap_or() or unwrap_or_else(). If the the Option is transformed into a Result the ? operator should be used as described above.

  Example:

  ```
  let checked_value = get_some_option().ok_or(SomeError)?;
  let checked_int = get_some_int_option().unwrap_or(42);
  ```

References

- The Rust Programming Language – Recoverable Errors with Result – Propagating errors: https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html#propagating-errors

3.3 Copy-paste code

Description

A lot of code seems to be copied which results in code duplication and possibly unneeded code. It becomes an issue when a bug is fixed but remains in one of the copies.

Proof of Issue

As an example, the 94 lines of code are identical in lib.rs in serum-amm/crank. The lines from 265 to 358 are identical to the lines from 1127 to 1216 with exception of line 330 and 331.

Filename: serum-amm/crank/src/lib.rs

Beginning Line Number: 262 / 1123

```
262.  pub fn load_config(opts_vec:
      &mut Vec<Opts>) -> Result<()>
      {
263.    let mut config = Ini::new();
264.    let _map =
      config.load("config.ini").unwr
      ap();
265.    let payer_file =
      config.get("Global",
      "payer_file").unwrap();
266.    if payer_file.is_empty() {
267.      panic!("payer_file must
      not be empty");
268.    }
269.    let cluster_url =
      config.get("Global",
      "cluster_url").unwrap();
270.    if cluster_url.is_empty() {
271.      panic!("cluster_url must
      not be empty");
272.    }
273.    let amm_program_str =
      config.get("Global",
      "amm_program_id").unwrap();
274.    let amm_program_id;
275.    if
      !amm_program_str.is_empty() {
276.      amm_program_id =
      Pubkey::from_str(&amm_program_
      str)?;
277.    }
278.    else {
279.      panic!("amm_program_id
      must not be empty");
280.    }
```

```
1123. fn
      auto_generate_amm_loop(config_
      path: &str)-> Result<()>
1124. {
1125.   let mut config = Ini::new();
1126.   let _map =
      config.load(config_path).unwra
      p();
1127.   let payer_file =
      config.get("Global",
      "payer_file").unwrap();
1128.   if payer_file.is_empty() {
1129.     panic!("payer_file must
      not be empty");
1130.   }
1131.   let cluster_url =
      config.get("Global",
      "cluster_url").unwrap();
1132.   if cluster_url.is_empty() {
1133.     panic!("cluster_url must
      not be empty");
1134.   }
1135.   let amm_program_str =
      config.get("Global",
      "amm_program_id").unwrap();
1136.   let amm_program_id;
1137.   if
      !amm_program_str.is_empty() {
1138.     amm_program_id =
      Pubkey::from_str(&amm_program_
      str)?;
1139.   }
1140.   else {
1141.     panic!("amm_program_id not
      provide");
1142.   }
```

Recommendation

Move abstractions of functionality into separate functions. In the above example the 94 lines can be moved to a function that takes a file names as parameter.

Furthermore, much of the configuration setup is about passing configuration options to creation certain structs. This can also be abstracted as they are highly identical functionality.

References

- Cargo Cult Programming: https://en.wikipedia.org/wiki/Cargo_cult_programming

3.4 Code cleaning

Description

The source files contain a lot of out-commented code. The lines that are ignored but takes up space and confuses other programmers about the intention of the code.

Proof of Issue

The following code snippet includes several lines of out-commented code. However, it is confusing whether the intension is to simulate the transaction before it is committed or if it is just some leftover code from an initial simulation implementation.

Filename: serum-amm/crank/src/lib.rs

Beginning Line Number: 1730

```
1730.   let txn = Transaction::new_signed_with_payer(
1731.           &instructions,
1732.           Some(&payer.pubkey()),
1733.           &signers,
1734.           recent_hash,
1735.       );
1736.
1737.       // println!("txn:\n{:#x?}", txn);
1738.       //  let  result  =  simulate_transaction(client,  &txn,  true,
        CommitmentConfig::single())?;
1739.       // if let Some(e) = result.value.err {
1740.       //        return  Err(format_err!("simulate_transaction  error:
        {:?}", e));
1741.       // }
1742.       // println!("{:#?}", result.value);
1743.
1744.       println!("Listing {} ...", market_key.pubkey());
1745.       send_txn(client, &txn, false)?;
```

Recommendation

Remove all out-commented code line.

3.5 Dead code

Description

Linters such as clippy will detect dead code. Dead code is code that will never get executed because it is unreachable.

However, the warnings have been disabled intentionally. This is not recommended as it creates an unnecessary larger code base to maintain.

Proof of Issue

Filename: serum-amm/crank/src/lib.rs

Beginning Line Number: 2

```
2.  #![allow(dead_code)]
```

Recommendation

1. Remove the allowance of dead code.
2. Remove all dead code.

4. TECHNICAL DETAILS

This section contains the technical details of our findings as well as recommendations for improvement.

4.1 Compiler warnings

Finding ID: KS-RAYDIUM-F-01

Severity: **Informational**

Status: **Open**

Description

The compiler generates warnings when building the code.

Proof of Issue

Output from cargo build for raydium-staking/crank shows 19 warnings: 15 warnings in the crank project and 4 warnings in the program project:

```
⋮
   Compiling raydium-staking v0.1.0 (/home/review/raydium-staking/program)
warning: unused import: `std::str::FromStr`
  --> /home/review/raydium-staking/program/src/processor.rs:30:5
```

```
⋮
warning: unused variable: `program_id`
   --> /home/review/raydium-staking/program/src/processor.rs:238:9
⋮
warning: unused variable: `program_id`
   --> /home/review/raydium-staking/program/src/processor.rs:340:9
⋮
warning: unused variable: `program_id`
   --> /home/review/raydium-staking/program/src/processor.rs:443:9
⋮
warning: 4 warnings emitted
   Compiling staking_crank v0.2.0 (/home/review/raydium-staking/crank)
warning: unused import: `mint_to_new_account`
  --> src/main.rs:23:77
⋮
warning: unused imports: `set_fee_owner`, `set_fee`
  --> src/main.rs:27:18
⋮
warning: unreachable expression
   --> src/main.rs:394:5
⋮
warning: unused variable: `map`
  --> src/main.rs:85:9
⋮
warning: variable `stake_pool_pk` is assigned to, but never used
   --> src/main.rs:331:21
⋮
warning: value assigned to `stake_pool_pk` is never read
   --> src/main.rs:333:21
⋮
warning: variable does not need to be mutable
   --> src/main.rs:157:9
⋮
warning: function is never used: `create_staker_info_account`
   --> src/main.rs:619:4
⋮
warning: function is never used: `warp_sol`
   --> src/main.rs:639:4
⋮
warning: function is never used: `unwarp_sol`
   --> src/main.rs:677:4
⋮
warning: function is never used: `deposit_txn`
   --> src/main.rs:739:4
⋮
warning: function is never used: `withdraw_txn`
   --> src/main.rs:858:4
⋮
warning: function is never used: `emergency_withdraw_txn`
   --> src/main.rs:889:4
⋮
```

```
warning: function is never used: `set_reward_param_txn`
    --> src/main.rs:1165:4
⋮
warning: unused `std::result::Result` that must be used
   --> src/main.rs:346:21
⋮

warning: 15 warnings emitted
    Finished dev [unoptimized + debuginfo] target(s) in 3m 19s
```

Severity and Impact Summary

Compiler warnings often indicate programming mistakes and should therefore be fixed at once.

From the proof of issue above the last warning in main.rs line 346 indicate that the Result returned from update_pools_txn() is never used. Here is the context:

Filename: raydium-staking/crank/src/main.rs

Beginning Line Number: 345

```
345.   loop {
346.       update_pools_txn(&pool, &client, &payer, &stake_pool_pks);
347.       std::thread::sleep(std::time::Duration::new(5, 0));
348.   }
```

If an error is returned by the call to update_pool_txn() it is never handled! The loop just continues as if nothing happened. Even if this is on purpose it is bad practice to ignore error handling.

This could have been spotted if all compiler warnings were eliminated during development.

Recommendation

Correct the code so the compiler does not produce any warnings.

It should always be a priority that the code can be compiled without errors or warnings. Compiling should be built into the continuous integration pipeline to automatically ensure high quality of the code.

4.2 Failed test

Finding ID: KS-RAYDIUM-F-02

Severity: **Low**

Status: **Open**

Description

When running tests for serum-amm/amm one of the tests fail.

Proof of Issue

Output from cargo test-bpf for serum-amm/amm shows that 1 out of 4 tests fail:

```
running 4 tests
test processor::srm_token::test_id ... ok
test processor::amm_owner::test_id ... ok
test processor::msrm_token::test_id ... ok
test instruction::tests::test_instruction_packing ... FAILED

failures:

---- instruction::tests::test_instruction_packing stdout ----
thread 'instruction::tests::test_instruction_packing' panicked at 'called
`Result::unwrap()` on an `Err` value: Custom(28)',
src/instruction.rs:1203:35
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace


failures:
    instruction::tests::test_instruction_packing

test result: FAILED. 3 passed; 1 failed; 0 ignored; 0 measured; 0 filtered
out; finished in 0.01s

error: test failed, to rerun pass '--lib'
```

The test instruction::tests::test_instruction_packing fails due to unwrapping of an Err result:

Filename: serum-amm/amm/src/instructions.rs

Beginning Line Number: 1201

```
1201.  let param: u8 = 7;
1202.  let check = AmmInstruction::SetParams(SetParamsInstruction{param, value:
    None, new_pubkey: Some(new_pubkey), fees: None});
1203.  let packed = check.pack().unwrap();
1204.  let mut expect = vec![6];
1205.  expect.push(param);
1206.  expect.extend_from_slice(&new_pubkey.to_bytes());
1207.  assert_eq!(packed, expect);
```

Err is returned by check.pack() that matches the SetParams on line 462. As param is 7 matches neither AmmParams::AmmOwner (10), AmmParams::PnlOwner (11), nor AmmParams::Fees (9) the underscore in line 482. As value is None the match returns an Err(AmmError::InvalidInput.into()) on line 485.

Filename: serum-amm/amm/src/instructions.rs

Beginning Line Number: 407

```
407.   pub fn pack(&self) -> Result<Vec<u8>, ProgramError> {
408.       let mut buf = Vec::with_capacity(size_of::<Self>());
409.       match &*self {
           // ...
462.       Self::SetParams(SetParamsInstruction{param, value, new_pubkey, fees})
    => {
463.           buf.push(6);
464.           buf.push(*param);
465.           match AmmParams::from_u64(*param as u64) {
466.               AmmParams::AmmOwner | AmmParams::PnlOwner => {
                       // ...
472.               },
473.               AmmParams::Fees => {
                       // ...
481.               },
482.               _ => {
483.                   let value = match value {
484.                       Some(a) => a,
485.                       None => return Err(AmmError::InvalidInput.into())
486.                   };
                       // ...
```

Reading a bit further in the test to line 1207 the test expects that the returned value from pack() is a vec containing 7 and the new_pubkey. So it seems like the functionality for packing a SetParamsInstruction with AmmParams::MinSize (7) has not been implemented yet…

Severity and Impact Summary

A failing test indicates that some functionality is broken. In this specific case it seems that the functionality that the test is verifying has not yet been implemented.

It should always be a priority that all tests pass. Automatic testing should be built into the continuous integration pipeline along with building to automatically ensure high quality of the code.

Recommendation

Implemented the missing functionality to make the test pass.

4.3 Low test coverage

Finding ID: KS-RAYDIUM-F-03

Severity: **Low**

Status: **Open**

Description

The test coverage is very low!

Proof of Issue

grcov has been used to calculate the test coverage of serum-amm/amm. The other projects contain no tests at all…

PROJECT	TEST COVERAGE
serum-amm/amm	9.1%
serum-amm/crank	0.0%
raydium-staking/program	0.0%
raydium-staking/crank	0.0%
Total	**4.1%**

Severity and Impact Summary

High test coverage does not necessarily ensure that the code works as expected. But low test coverage indicates that much of the code has not been verified at all.

Only 4.1% of the written code has been tested. This is far from convincing when it comes to verifying that intension of the written code is correct…

The low coverage can introduce a security issues in the long run as it makes the code untestable and unmaintainable. Thus, the maintainers will not be able to rely on the intention of the code or be sure that even simple updates do not break functionality. And if updates cannot be implemented reliantly then it becomes a high security concern!

Furthermore, software security is needless if the software does not behave as expected to begin with.

Recommendation

Implement unit tests and integration tests to verify the intention of the code. This will make a base for regression testing ensuring that functionality has not been broken when future updates such as security patches are implemented in the future.

References

- grcov – Generate a coverage report from coverage artifacts: https://github.com/mozilla/grcov#generate-a-coverage-report-from-coverage-artifacts

4.4 Code formatting

Finding ID: KS-RAYDIUM-F-04

Severity: **Informational**

Status: **Open**

Description

Code that does not follow a set of standard formatting rules is analogous to reading a novel without punctuation characters. The words of the novel can still be read but the reader will be left on his own to figure out when sentences start and stop.

In short, not following a set of standard formatting rules makes code harder to read and understand by other developers than the author. Hence, it creates a risk if other developers should be able the maintain the code.

Complying to the Rust formatting code style can easily be achieved by using rustfmt command (alias cargo fmt). However, rustfmt has not been applied to format the code according to the Rust community style guidelines…

Proof of Issue

`cargo fmt` suggests formatting changes for:

PROJECT	SUGGESTED STYLE CHANGES
serum-amm/amm	825 lines
serum-amm/crank	814 lines
raydium-staking/program	155 lines
raydium-staking/crank	270 lines
Total	**2,064 lines**

Severity and Impact Summary

Not following a set of standard formatting rules makes code harder to read and understand. Hence, it creates a maintenance risk.

Recommendation

Review and apply code formatting changes from rustfmt as a part of the development process.

The code format check should be built into the continuous integration pipeline along with building and testing to automatically ensure high quality of the code.

References

- Rust Style Guide: https://github.com/rust-dev-tools/fmt-rfcs/blob/master/guide/guide.md
- rustfmt – Checking style on a CI server: https://github.com/rust-lang/rustfmt#checking-style-on-a-ci-server

4.5 Bad code practice

Finding ID: KS-RAYDIUM-F-05

Severity: **Informational**

Status: **Open**

Description

A linter is a tool that inspects the code for bad code practices of language constructs that lead to programming errors or incomprehensive code. In Rust, the clippy command implements more than 400 lint inspections for bad code practices.

However, clippy has not been applied to indicate common code errors…

Proof of Issue

cargo clippy has been used to check for bad code practices. The following number of errors and warnings were found:

PROJECT	ERRORS	WARNINGS
serum-amm/amm*	2	42
serum-amm/crank*	2	40
raydium-staking/program	0	11
raydium-staking/crank	0	37
Total	**4**	**130**

*: It must be noted that the errors and warnings found in serum-amm are from the dependencies to serum-dex which is out of scope of this review. However, there are still errors and warning in serum-dex that must be fixed!

An example of an error from serum-dex:

```
error: this loop never actually loops
   --> /home/review/raydium-amm/serum-dex/dex/src/matching.rs:332:20
    |
332 |           let done = loop {
    |  ____________________^
333 | |             let best_bid_h = match self.find_bbo(Side::Bid) {
334 | |                 None => {
335 | |                     crossed = false;
...   |
474 | |             break false;
475 | |         };
    | |_________^
    |
    = note: `#[deny(clippy::never_loop)]` on by default
    = help: for further information visit https://rust-lang.github.io/rust
-clippy/master/index.html#never_loop
```

It can be hard to spot as the loop block is 143 lines. But the linter detects that last line in the loop block always breaks the loop (and there are not continues). So, using a loop is quite misguiding.

To be able to break out of the block it could be refactored out as another method changing the breaks into returns. But a more radical refactoring is recommended to split the many lines into smaller abstractions that are easier to comprehend.

An example of a warning from raydium-staking/program:

```
warning: casting integer literal to `u128` is unnecessary
   --> /home/review/raydium-staking/program/src/processor.rs:306:30
    |
306 |                 .checked_div(1_000_000_000  as u128).unwrap()
    |                              ^^^^^^^^^^^^^^^^^^^^^^
    |                              help: try: `1_000_000_000_u128`
    |
    = note: `#[warn(clippy::unnecessary_cast)]` on by default
    = help: for further information visit https://rust-lang.github.io/rust
-clippy/master/index.html#unnecessary_cast
```

It is less cryptical to define an unsigned 128 bit integer directly than defining a signed 32 bit integer and casting it to an unsigned 128 bit integer.

Severity and Impact Summary

Bad code practices can lead to either programming errors or incomprehensive code. Programming errors introduce integrity risks and incomprehensive code introduces a maintenance risk.

Recommendation

It should always be a priority to remove all errors and warnings from indicated by the linters.

Linting should be built into the continuous integration pipeline along with building, testing, and code format checking to automatically ensure high quality of the code.

References

- Clippy:
  https://github.com/rust-lang/rust-clippy

4.6 Vulnerabilities from RustSec Advisory Database

Finding ID: KS-RAYDIUM-F-06

Severity: **Low**

Status: **Open**

Description

Dependencies should automatically be checked for known security issues registered in the RustSec Advisory Database.

Proof of Issue

cargo audit has been used to check for known security issues. The following number of errors and warnings were found:

PROJECT	ERRORS	WARNINGS
serum-amm/amm	1	0
serum-amm/crank	0	0
raydium-staking/program	2	2
raydium-staking/crank	2	5
Total	**5**	**7**

The errors are listed below:

Projects	serum-amm/amm raydium-staking/program raydium-staking/crank
Dependency	generic-array 0.12.3
Title	arr! macro erases lifetimes
Description	Affected versions of this crate allowed unsoundly extending lifetimes using arr! macro. This may result in a variety of memory corruption scenarios, most likely use-after-free.
Reference	https://rustsec.org/advisories/RUSTSEC-2020-0146
Solution	Upgrade to >=0.8.4, <0.9.0 OR >=0.9.1, <0.10.0 OR >=0.10.1, <0.11.0 OR >=0.11.2, <0.12.0 OR >=0.12.4, <0.13.0 OR >=0.13.3

Projects	raydium-staking/program raydium-staking/crank
Dependency	rand_core 0.6.1
Title	Incorrect check on buffer length when seeding RNGs
Description	Summary: rand_core::le::read_u32_into and read_u64_into have incorrect checks on the source buffer length, allowing the destination buffer to be under-filled. Implications: some downstream RNGs, including Hc128Rng (but not the more widely used ChaCha*Rng), allow seeding using the SeedableRng::from_seed trait-function with too short keys.
Reference	https://rustsec.org/advisories/RUSTSEC-2021-0023
Solution	Upgrade to >=0.6.2

The warnings indicate dependencies that are unmaintained and deprecated for use.

Severity and Impact Summary

The two errors found may impact:

- memory corruption
- seeding of random generators with too short keys

Recommendation

It should always be a priority to remove all errors and warnings from the security audit.

Review and apply audit solutions from the RustSec Advisory Database as a part of the development process.

The security audit should be built into the continuous integration pipeline along with building, testing, code format checking, and linting to automatically ensure high quality of the code.

References

- The RustSec Advisory Database: https://rustsec.org

4.7 Zero out sensitive data

Finding ID: KS-RAYDIUM-F-07

Severity: **Medium**

Status: **Open**

Description

When deallocating memory, the reservation of the memory block is released so it can be used by other processes. However, the data that written to the memory block remain. Another process may allocate the memory block containing the previously used data and read that data. This is a security risk if the previously used data is of sensitive nature!

Sensitive data used in the serum-amm/crank project includes keypairs. A keypair consists of a public and a private key. The private key contains sensitive information that should not be leaked.

The Solana SDK is used to read keypairs from files and write keypairs to files. However, the Solana SDK does nothing to ensure that the keypair and especially the private key is erased from memory when it is deallocated!

So other processes may acquire a used private key by accessing the deallocated memory block used for a private key during execution of the program… This is a security risk!

For this issue to be exploited, not only would there have to be the right sequence of events to put another private key at this memory location, but an attacker would need access to the environment OR the local swap where this information is stored. It is likely that a cloud system or an operating system would stop a non-administrative user from abusing this. The resultant risk of this scenario is medium.

The following function calls lead to use of keypairs where memory of the read private key must be zeroed out after use:

FUNCTION CALL	MODULE	OCCURRENCES
read_keypair_file()	serum-amm/crank	49
read_keypair_file()	raydium-staking/crank	5
Keypair::from_bytes()	serum-amm/crank	3
Keypair::generate()	serum-amm/crank	9
Keypair::generate()	raydium-staking/crank	12

Proof of Issue

In the code snippet below the payer keypair is read from a file. But when the block ends the keypair is just deallocated without zeroing out the sensitive data of the private key.

Filename: serum-amm/crank/src/lib.rs

Beginning Line Number: 677

```
677.  Command::GenerateTokenMint {
678.      ref payer_path,
679.      coin_decimals,
680.      pc_decimals,
681.  } => {
682.      let client = opts.client();
683.      let payer = read_keypair_file(payer_path)?;
684.      let token_mints = generate_mint(&client, &payer, coin_decimals,
      pc_decimals).unwrap();
685.      println!("{:#?}", token_mints);
686.  }
```

Severity and Impact Summary

Other processes may acquire a used private key by accessing the deallocated memory areas used for a private key during execution of the program… This is a security risk!

Recommendation

To avoid access to private keys in deallocated memory the memory area holding the private key must be zeroed out as soon as the private key is not needed in scope.

It is important that zeroing out is called even if panic or errors occur! Therefore, an automated mechanism such as zeroize should be used.

References

- Zeroize:
  https://docs.rs/zeroize

4.8 Conditional compilation for little endian only

Finding ID: KS-RAYDIUM-F-08

Severity: **Low**

Status: **Open**

Description

Conditional compilation for little endian is used various times in the code. This might be intentionally but seems more like a copy-paste-programming…

The limitation to little endian seems to be used when the safe transmute library to serialize/deserialize between raw bytes and structs or initializing structs with zeroed data.

It is convenient to use transmute to serialize data structures into raw bytes. But it also brings several pitfalls including handling endianness, memory layout, memory alignment, and type limitations. To cite The Rustonomicon on Transmutes:

> *Even though this book is all about doing things that are unsafe, I really can't emphasize that you should deeply think about finding Another Way than the operations covered in this section. This is really, truly, the most horribly unsafe thing you can do in Rust.*
>
> *– The Rustonomicon, Transmutes*

An example of a better approach to serialize data is used in pack() and unpack() of the AmmInstruction struct in serum-amm/amm/src/instruction.rs. This implementation makes explicit use of from_le_bytes() and to_le_bytes() to ensure that the conversion always respects the required endianness and substantially less unsafe code.

Proof of Issue

The serum_amm::state module alone contains 15 lines with conditional compilation configurations for little endian.

Filename: serum-amm/amm/src/state.rs

Beginning Line Number: 36

```
36.  #[cfg_attr(feature = "client", derive(Debug))]
37.  #[derive(Clone, Copy)]
38.  pub struct WithdrawQueue {
```

```
39.       owner: [u64;4],
40.       head: u64,
41.       count: u64,
42.       buf: [WithdrawDestToken; 64],
43.   }
44.
45.   #[cfg(target_endian = "little")]
46.   unsafe impl Zeroable for WithdrawQueue {}
47.   #[cfg(target_endian = "little")]
48.   unsafe impl Pod for WithdrawQueue {}
49.   #[cfg(target_endian = "little")]
50.   unsafe impl TriviallyTransmutable for WithdrawQueue {}
```

Although the conditional configurations will make the compiler ignore only the specified code, it will imply that the whole project will not compile.

For example, if compiled for big endian the Zeroable trait will not be implemented for the WithdrawQueue struct in line 46 in the code above. This will result in a compilation error in serum-amm/crank where Zeroable::zeroed() is called.

Filename: serum-amm/crank/src/lib.rs

Beginning Line Number: 2190

```
•     let mut withdraw_queue:WithdrawQueue = Zeroable::zeroed();
```

Conditional compilation configuration is also used in:

FILE	NUMBER OF CONDITIONAL ENDIANNESS
serum-amm/amm/src/state.rs	15
serum-amm/crank/src/lib.rs	2
raydium-staking/program/src/state.rs	6
Total	**23**

Severity and Impact Summary

Currently, the code will only compile for little endian architectures which is the most common architecture today.

The 23 conditional configurations for compiling only on little endian target architectures adds unnecessary complexity to the code.

The conditional configurations seem to be used to ensure that serialization of data structures can be done using transmute. This brings adds further complexity and pitfalls such as handling endianness, memory layout, memory alignment, and type limitations.

Recommendation

Limit conditional compilation configuration to only appear where actually needed. Furthermore, it is best practice to implement code for both endian cases. Implement code for big endian architecture where needed.

If conditional compilation is kept, testing should be done on both little and big endian architecture. This can easily be achieved using the “Zero setup cross compilation and cross testing of Rust crates” with MIPS as target (see references).

Implement serialization and deserialization of data structures manually instead of using unsafe traits from safe_transmute such as TriviallyTransmutable and Pod (see AmmInstruction::pack() and unpack() for inspiration).

Implement zeroed initialization as constructor functions instead of using unsafe traits from bytemuck such as Zeroable.

References

- The Rustonomicon, Transmutes: https://doc.rust-lang.org/nomicon/transmutes.html
- Zero setup cross compilation and cross testing of Rust crates: https://github.com/rust-embedded/cross

4.9 Invalid reference type

Finding ID: KS-RAYDIUM-F-09

Severity: **Low**

Status: **Open**

Description

Values of different types are inserted into a vector.

Proof of Issue

The following code should create signers of type Vec<&Keypair>. But the inserted values are payer of type &Keypair and &token_account of type &&Keypair!

Fortunately, the compiler fixes this inconsistency, but it can easily become a problem when updating the code…

Filename: serum-amm/crank/src/lib.rs

Beginning Line Number: 1877

```
1877.  fn init_token_account(
1878.      client: &RpcClient,
1879.      token_account: &Keypair,
1880.      mint_pubkey: &Pubkey,
1881.      owner_pubkey: &Pubkey,
1882.      payer: &Keypair,
1883.  ) -> Result<()> {
           // ...
1903.      let signers = vec![payer, &token_account];
```

Invalid reference types appear in:

FILE	LINES
serum-amm/crank/src/lib.rs	1877
raydium-staking/crank/src/main.rs	758, 877, 905, 974, 1038, 1099, 1190

Severity and Impact Summary

This reference type error is fixed by the compiler.

But it is intentionally and removing by not inserting the first value into the vector would change the type and the following behavior of the code.

Recommendation

Update the code so the second value is not of type &&Keypair:

```
let signers = vec![payer, token_account];
```

4.10 Fibonacci integer overflow

Finding ID: KS-RAYDIUM-F-10

Severity: **Medium**

Status: **Open**

Description

The implementation of Fibonacci does not check for overflow. When storing values as unsigned 64 bit integers the largest Fibonacci number that can be stored is Fibonacci(93). Fibonacci(94) will cause an integer overflow.

VALUE NAME	INTEGER VALUE
Fibonacci(93)	12,200,160,415,121,876,738
Max. unsigned 64 bit integer	18,446,744,073,709,551,615
Fibonacci(94)	19,740,274,219,868,223,167

Proof of Issue

Filename: serum-amm/amm/src/math.rs

Beginning Line Number: 113

```
113.  pub fn fibonacci(order_num:u64) -> Vec<u64> {
114.    let mut fb = Vec::new();
115.    for i in 0..order_num {
116.      if i == 0 {
117.        fb.push(0u64);
118.      } else if i == 1 {
119.        fb.push(1u64);
120.      } else if i == 2 {
121.        fb.push(2u64);
122.      } else {
123.        let ret = fb[(i - 1u64) as usize] +fb[(i - 2u64) as usize];
124.        fb.push(ret);
125.      };
126.    }
127.    return fb;
128.  }
```

On line 123 an unchecked add is used which will cause an overflow when order_num is 93. The order_num comes from the AmmInfo struct derived from the accounts data handed to the Solana entrypoint.

Severity and Impact Summary

In general, this opens for integer overflow attacks.

More specifically, the Fibonacci numbers are used in serum-amm/amm to create buying plans. If an integer overflow occurs it may lead to a miscalculation of the buying price.

Recommendation

Implement verification that the order_num is less than or equal to 93.

Wrap the integer addition in the Fibonacci function and make the function return a Result. This way an error can be returned on integer overflows.

4.11 Unchecked power functions

Finding ID: KS-RAYDIUM-F-11

Severity: **Medium**

Status: **Open**

Description

Although checked calls to the mathematical functions are used widely in math.rs in serum-amm/amm, the power function has been left unchecked.

Proof of Issue

Filename: serum-amm/amm/src/math.rs

Beginning Line Number: 130

```
130.  pub fn normalize_decimal(val: u64, native_decimal: u64,
      sys_decimal_value: u64) -> u64 {
131.    // e.g., amm.sys_decimal_value is 10**6, native_decimal is 10**9,
      price is 1.23, this function will convert (1.23*10**9) ->
      (1.23*10**6)
132.    //let ret:u64 =
      val.checked_mul(amm.sys_decimal_value).unwrap().checked_div((10 as
      u64).pow(native_decimal.into())).unwrap();
133.    let ret_mut:u128 = (val as
      u128).checked_mul(sys_decimal_value.into()).unwrap();
134.    let ret = Self::to_u64(ret_mut.checked_div((10 as
      u128).pow(native_decimal.try_into().unwrap())).unwrap()).unwrap();
135.    ret
136.  }
```

In the code above native_decimal is used as exponent to raise the base 10.

When storing values as unsigned 128 bit integers the largest exponent that can be used for the power of base 10 is 38 as $10^n < 2^{128} \Leftrightarrow n < \log_{10}(2^{128}) = 38.5$.

VALUE NAME	INTEGER VALUE
10^{38}	100,000,000,000,000,000,000,000,000,000,000,000
Max. unsigned 128 bit integer	340,282,366,920,938,463,463,374,607,431,768,211,455
10^{39}	1,000,000,000,000,000,000,000,000,000,000,000,000

Note that the call to to_u64() on line 134 will cause panic if the division result is larger than the max. value of an unsigned 64 bit integer. But this will not catch overflows of the power function. Instead, this will cause an invalid calculation.

The same issue goes for other options:

NAME	FILE NAME	LINES
native_decimal	serum-amm/amm/src/math.rs	133, 139
pc_decimals	serum-amm/amm/src/math.rs	169, 180
coin_decimals	serum-amm/amm/src/math.rs	170, 178, 203, 211

Severity and Impact Summary

In general, this opens for integer overflow attacks through the native_decimals, pc_decimals, and coin_decimals values.

Some verifications are implemented but these do not catch all overflows which will lead to unexpected behavior…

Recommendation

Implement verification that native_decimals, pc_decimals and coin_decimals are less than 39.

Use the checked_pow() function to detect overflow when using a power function.

References

- Function num_traits::pow::checked_pow: https://docs.rs/num-traits/0.2.14/num_traits/pow/fn.checked_pow.html

4.12 Unchecked conversion from internal pc lot size to srm pc lot size

Finding ID: KS-RAYDIUM-F-12

Severity: **Medium**

Status: **Open**

Description

Although checked calls to the mathematical functions are used widely in math.rs in serum-amm/amm, the implementation of convert_out_pc_lot_size() has been left unchecked.

Proof of Issue

Filename: serum-amm/amm/src/math.rs

Beginning Line Number: 168

```
168.  pub fn convert_out_pc_lot_size(pc_decimals: u8, coin_decimals: u8,
      pc_lot_size: u64, coin_lot_size: u64, sys_decimal_value: u64) ->
      u64 {
169.    let native_lot_size = Self::to_u64(((pc_lot_size as u128) *
      (coin_lot_size as u128) * ((10 as u128).pow(pc_decimals.into()))) /
170.      ((sys_decimal_value as u128) * ((10 as
      u128).pow(coin_decimals.into())))).unwrap();
171.    native_lot_size
172.  }
```

The above code contains 3 unchecked multiplications and an unchecked division. This will integer overflow when either:

- $pc\_lot\_size \cdot coin\_lot\_size \cdot 10^{pc\_decimals} < 2^{128}$
- $sys\_decimal\_value \cdot 10^{coin\_decimals} < 2^{128}$
- $10^{pc\_decimals} < 2^{128} \Leftrightarrow pc\_decimals < \log_{10}(2^{128}) < 39$
- $10^{coin\_decimals} < 2^{128} \Leftrightarrow coin\_decimals < \log_{10}(2^{128}) < 39$

Furthermore, the formula should be reduced to decrease the exponent of the power function making it more robust against integer overflows.

$$\frac{pc\_lot\_size \cdot coin\_lot\_size \cdot 10^{pc\_decimals}}{sys\_decimal\_value \cdot 10^{coin\_decimals}}$$

$$= \begin{cases} \dfrac{pc\_lot\_size \cdot coin\_lot\_size}{sys\_decimal\_value \cdot 10^{coin\_decimals - pc\_decimals}}, & \text{pc\_decimals} < \text{coin\_decimals} \\ \dfrac{pc\_lot\_size \cdot coin\_lot\_size \cdot 10^{pc\_decimals - coin\_decimals}}{sys\_decimal\_value}, & \text{pc\_decimals} \geq \text{coin\_decimals} \end{cases}$$

The limits are then:

- $|pc\_decimals - coin\_decimals| < 39$
- $pc\_lot\_size \cdot coin\_lot\_size \cdot 10^{|pc\_decimals - coin\_decimals|} < 2^{128}$
- $sys\_decimal\_value \cdot 10^{|pc\_decimals - coin\_decimals|} < 2^{128}$

Severity and Impact Summary

In general, this opens for integer overflow attacks through the pc_decimals, pc_lot_size, coin_decimals, and coin_lot_size values.

Recommendation

Use the checked_mul(), checked_div(), and checked_pow() function to detect overflow when using a power function.

Decrease the power exponent with mathematical reduction to make the implementation more robust against integer overflows.

APPENDIX A: ABOUT KUDELSKI SECURITY

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com.

Kudelski Security

route de Genève, 22-24

1033 Cheseaux-sur-Lausanne

Switzerland

Kudelski Security

5090 North 40th Street

Suite 450

Phoenix, Arizona 85018

APPENDIX B: DOCUMENT HISTORY

VERSION	STATUS	DATE	AUTHOR	COMMENTS
0.1	Draft	14 April 2021	Scott Carlson	First draft
0.2	Draft	16 April 2021	Mikael Björn	Draft to QA
1.0	Final Draft	29 April 2021	Scott Carlson	Final for Review
1.0	Final Public	11 May 2021	Scott Carlson	Final

APPENDIX C: SEVERITY RATING DEFINITIONS

Kudelski Security uses a custom approach when determining criticality of identified issues. This is meant to be simple and fast, providing customers with a quick at a glance view of the risk an issue poses to the system. As with anything risk related, these findings are situational. We consider multiple factors when assigning a severity level to an identified vulnerability. A few of these include:

- Impact of exploitation
- Ease of exploitation
- Likelihood of attack
- Exposure of attack surface
- Number of instances of identified vulnerability
- Availability of tools and exploits

SEVERITY	DEFINITION
High	The identified issue may be directly exploitable causing an immediate negative impact on the users, data, and availability of the system for multiple users.
Medium	The identified issue is not directly exploitable but combined with other vulnerabilities may allow for exploitation of the system or exploitation may affect singular users. These findings may also increase in severity in the future as techniques evolve.
Low	The identified issue is not directly exploitable but raises the attack surface of the system. This may be through leaking information that an attacker can use to increase the accuracy of their attacks.
Informational	Informational findings are best practice steps that can be used to harden the application and improve processes.