Competitive Security Assessment

ZKBase_Update

May 30th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

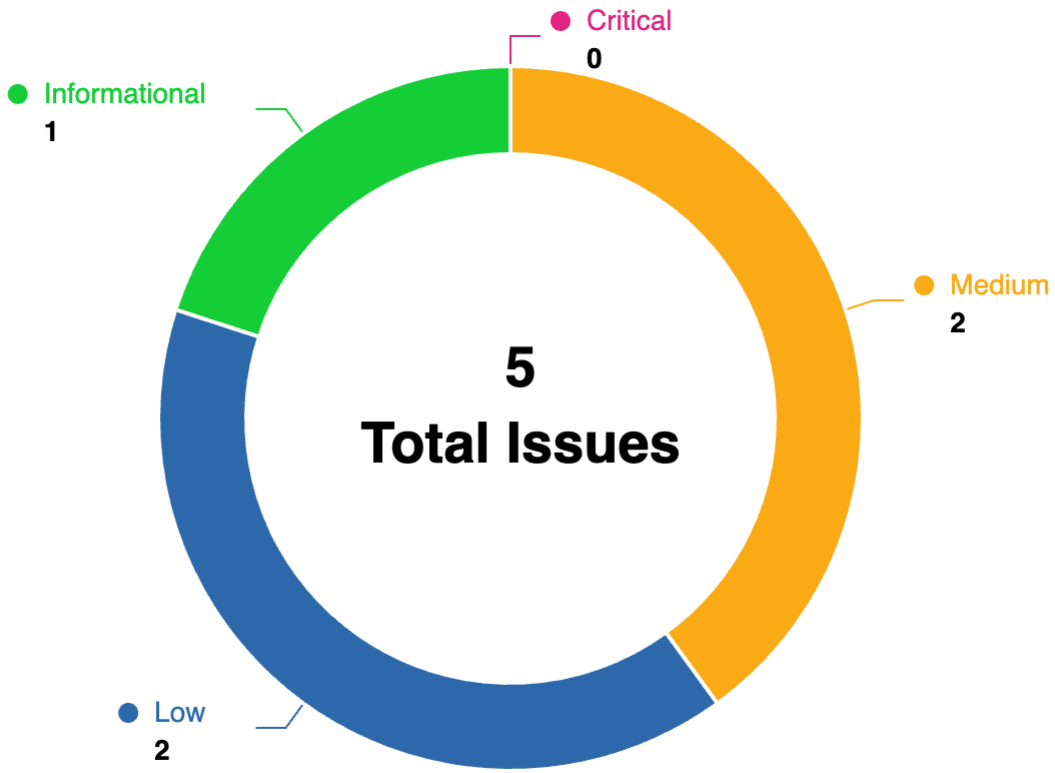
Overview

Project Name	ZKBase_Update
Language	Rust
Codebase	• https://github.com/l2labs/zksync-era.git • audit version - 9e400eaf29ace9e3e43a2eff11f20f4a471ee550 • final version - bfe2e439bfb4c900d443386b852cab6f0c78220e
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
core/lib/dal/src/proofs_dal.rs	5b7ab72b6e7d631d897811557b8d086a9f42ba8454b8aad1966d7f97de2a38f2
core/lib/object_store/src/pg.rs	bf9a81fc836d3ed6edf1534d04583ac0f99f9c48d945e2e88e1c8a27bfcf9f3f

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
ZSU-1	Vulnerability in Filename Generation Leads to Potential Data Overwrite	Logical	Medium	Fixed	yekong
ZSU-2	Inaccurate Data Deletion in del_proof Function Using LIKE Operator	Logical	Medium	Fixed	yekong, 0xac, ethprinter
ZSU-3	Potential DOS	DOS	Low	Acknowledged	Bryce
ZSU-4	No error to handle by `map_err`	Logical	Low	Acknowledged	Yaodao
ZSU-5	Potential `panic` in function `put_raw()`	Logical	Informational	Acknowledged	Yaodao

ZSU-1:Vulnerability in Filename Generation Leads to Potential Data Overwrite

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	yekong

Code Reference

- code/core/lib/object_store/src/pg.rs#L34-L92

```
34: fn filename(&self, bucket: Bucket, key: &str) -> String {
35:         format!("{bucket}_{key}")
36:     }
37: }
38:
39: #[async_trait]
40: impl ObjectStore for PGObjectStore {
41:     async fn get_raw(&self, bucket: Bucket, key: &str) -> Result<Vec<u8>, ObjectStoreError> {
42:         let filename = self.filename(bucket, key);
43:         println!("get bucket:{}, key:{}, finename:{}", bucket, key, filename);
44:
45:         let mut storage_processor = self.connection_pool.access_storage().await.unwrap();
46:
47:         let mut transaction = storage_processor.start_transaction().await.unwrap();
48:
49:         let ret = transaction.proof_dal().get_proof(filename).await;
50:
51:         transaction.commit().await.unwrap();
52:         Ok(ret).map_err(|e| ObjectStoreError::from(e))
53:     }
54:
55:     async fn put_raw(
56:         &self,
57:         bucket: Bucket,
58:         key: &str,
59:         value: Vec<u8>,
60:     ) -> Result<(), ObjectStoreError> {
61:         let filename = self.filename(bucket, key);
62:         println!("put bucket:{}, key:{}, finename:{}", bucket, key, filename);
63:
64:         let mut storage_processor = self.connection_pool.access_storage().await.unwrap();
65:
66:         let mut transaction = storage_processor.start_transaction().await.unwrap();
67:
68:         transaction
69:             .proof_dal()
70:             .insert_proof(filename, value, "lucy".to_string())
71:             .await;
72:
73:         transaction.commit().await.unwrap();
74:         Ok(()).map_err(|e| ObjectStoreError::from(e))
75:     }
76:
77:     async fn remove_raw(&self, bucket: Bucket, key: &str) -> Result<(), ObjectStoreError> {
78:         let filename = self.filename(bucket, key);
79:
80:         let mut storage_processor = self.connection_pool.access_storage().await.unwrap();
81:         let mut transaction = storage_processor.start_transaction().await.unwrap();
82:
83:         transaction.proof_dal().del_proof(filename).await;
84:
85:         transaction.commit().await.unwrap();
86:         Ok(()).map_err(|e| ObjectStoreError::from(e))
87:     }
88:
89:     fn storage_prefix_raw(&self, bucket: Bucket) -> String {
90:         format!("{}", bucket)
91:     }
92: }
```

Description

yekong: The `fn filename` function in the provided code base concatenates the `bucket` parameter and the `key` string using an underscore as a separator to generate filenames for data storage. This method does not ensure the uniqueness of the resulting filenames. For instance, `bucket` values of "public_docs" combined with a `key` of "2023_report" and `bucket` of "public" with a `key` of "docs_2023_report" both result in the same filename "public_docs_2023_report". This overlapping can lead to data being unintentionally overwritten in the `put_raw` and `remove_raw` function, where the generated filename is used directly to store data without checking for pre-existing filenames. This represents a critical vulnerability as there is no safeguard against filename collisions, which could result in data loss and compromise the integrity of stored data.

Code Snippet Illustration:

```
fn filename(&self, bucket: Bucket, key: &str) -> String {
    format!("{bucket}_{key}")
}

fn put_raw(
    &self,
    bucket: Bucket,
    key: &str,
    value: Vec<u8>,
) -> Result<(), ObjectStoreError> {
    let filename = self.filename(bucket, key);
    println!("put bucket:{}, key:{}, filename:{}", bucket, key, filename);

    let mut storage_processor = self.connection_pool.access_storage().await.unwrap();

    let mut transaction = storage_processor.start_transaction().await.unwrap();

    transaction
        .proof_dal()
        .insert_proof(filename, value, "lucy".to_string())
        .await;

    transaction.commit().await.unwrap();
    Ok(())
}
```

Recommendation

yekong: To mitigate this vulnerability, the system should adopt a more robust method for generating unique filenames. Implementing a hashing mechanism would be beneficial, as it would nearly eliminate the possibility of filename collisions. Here is an example of how the `fn filename` function could be modified to use hashing:

Suggested Code Modification:

```rust
use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;

fn filename(&self, bucket: Bucket, key: &str) -> String {
    let mut hasher = DefaultHasher::new();
    format!("{bucket}_{key}").hash(&mut hasher);
    format!("{}_{}", bucket, hasher.finish())
}

fn put_raw(
    &self,
    bucket: Bucket,
    key: &str,
    value: Vec<u8>,
) -> Result<(), ObjectStoreError> {
    let filename = self.filename(bucket, key);
    println!("put bucket:{}, key:{}, filename:{}", bucket, key, filename);

    let mut storage_processor = self.connection_pool.access_storage().await.unwrap();

    let mut transaction = storage_processor.start_transaction().await.unwrap();

    // Check if filename exists before writing data
    if transaction.proof_dal().filename_exists(filename).await? {
        return Err(ObjectStoreError::FilenameConflict);
    }

    transaction
        .proof_dal()
        .insert_proof(filename, value, "lucy".to_string())
        .await;

    transaction.commit().await.unwrap();
    Ok(())
}
```

Client Response

client response for yekong: Fixed. fixed in 8be825fa9bf2c56ce684ce4f3d172c8630327036.
We decide to panic the error where there is invalid character in parameter "key".

ZSU-2:Inaccurate Data Deletion in del_proof Function Using LIKE Operator

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	yekong, 0xac, ethprinter

Code Reference

- code/core/lib/dal/src/proofs_dal.rs#L59-L77
- code/core/lib/dal/src/proofs_dal.rs#L59-L76

```
59: pub async fn del_proof(
60:         &mut self,
61:         proof_name:String,
62:     ){
63:         sqlx::query!(
64:             r#"
65:             DELETE FROM
66:                 proofs
67:             WHERE
68:                 proof_name like $1
69:             "#,
70:             proof_name,
71:         )
72:         .execute(self.storage.conn())
73:         .await
74:         .unwrap();
75:     }
76: }
```


```
59: pub async fn del_proof(
60:         &mut self,
61:         proof_name:String,
62:     ){
63:         sqlx::query!(
64:             r#"
65:             DELETE FROM
66:                 proofs
67:             WHERE
68:                 proof_name like $1
69:             "#,
70:             proof_name,
71:         )
72:         .execute(self.storage.conn())
73:         .await
74:         .unwrap();
75:     }
76: }
```


- code/core/lib/object_store/src/pg.rs#L77

```
77: async fn remove_raw(&self, bucket: Bucket, key: &str) -> Result<(), ObjectStoreError> {
```

Description

yekong: The `PGObjectStore` class within the provided Rust codebase is designed to interact with a PostgreSQL database for storing and retrieving data based on unique identifiers (composed of `bucket` and `key`). However, there exists a potential SQL wildcard vulnerability due to the usage of the `LIKE` operator within the `proof_dal().get_proof` and `proof_dal().del_proof` methods. These methods are called in `get_raw` and `remove_raw` respectively with filenames generated via the `filename` method, which concatenates `bucket` and `key` with an underscore.
The vulnerability stems from how the filename is constructed and utilized in SQL queries. If the `key` parameter contains SQL wildcard characters such as `%` or `_`, the filename passed to SQL queries using the `LIKE` operator would lead to unintended broad pattern matching. This could result in erroneous data retrieval or deletion, where multiple database entries might be matched and manipulated instead of the targeted single entry.
0xac: A security review of the `del_proof` function within our application's codebase has revealed a significant risk associated with the use of the `LIKE` SQL operator for matching the `proof_name` parameter in deletion queries. This issue arises when the `proof_name` parameter contains SQL wildcard characters such as % or _, which are not intended for pattern matching but are treated as such by the SQL engine.
Issue Details:
The `del_proof` function is designed to delete entries from the proofs table based on a provided proof name. However, the function utilizes the `LIKE` operator in its SQL query, making it susceptible to unintentional pattern matching. For example, if a user or a process passes proof_name as "example%", the SQL query will inadvertently match and delete all entries where the proof names start with "example" (e.g., "example1", "exampleTest", etc.), not just an exact match.
Proof of Concept (POC):
Insert test data into proofs table:
`INSERT INTO proofs (proof_name) VALUES ('example'), ('example1'), ('exampleTest');`
Execute `del_proof` function with `proof_name = "example%"`:
All entries with proof names starting with "example" will be deleted, not just "example".
Impact:
This behavior can lead to unintended data loss, where records that should not be deleted are removed from the database. This compromises data integrity and can potentially impact application functionality, user data, and system reliability.
ethprinter: In `remove_raw()` function, it accepts 2 arguments `key` and `bucket`, both of them will be formatted to a string in `format!("{bucket}_{key}")`, after that the formatted string will be passed into `del_proof()` function.
`del_proof()` funtion will execute a sql statement as `DELETE FROM proofs WHERE proof_name like bucket_key`, the problem is if a user pass a wildcard like '%' into `remove_raw()`, the final sql string will be `DELETE FROM proofs WHERE proof_name like %_%`, so all the data in `proofs` will be deleted.

Recommendation

yekong: 1. **Adjust `proof_dal().get_proof` and `proof_dal().del_proof` methods:**
Modify these methods to use the `=` operator instead of `LIKE` for database operations. Here's an example modification for the `get_proof` method (similar changes should be applied to `del_proof`):

```
// Before: Using LIKE operator
let sql = "SELECT proof FROM proofs WHERE proof_name LIKE $1 LIMIT 1";

// After: Using equality operator
let sql = "SELECT proof FROM proofs WHERE proof_name = $1 LIMIT 1";
```

2. **Sanitize Input in `filename` Method:**
   Add input validation to the `filename` method to check for and reject or escape wildcard characters:

```
impl PGObjectStore {
    fn filename(&self, bucket: Bucket, key: &str) -> String {
        if key.contains('%') || key.contains('_') {
            // Log the error or handle it as per application requirements
            panic!("Input key contains invalid characters: {}", key);
        }
        format!("{bucket}_{key}")
    }
}
```

0xac: To mitigate this vulnerability and prevent potential data loss, it is recommended to modify the SQL query in the del_proof function to use the equals = operator for exact matching instead of the LIKE operator. This change ensures that only records with a proof name exactly matching the input parameter are deleted.
Code Fix:
Modify the del_proof function as follows:

```
pub async fn del_proof(
    &mut self,
    proof_name: String,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
        DELETE FROM
            proofs
        WHERE
            proof_name = $1
        "#,
        proof_name,
    )
    .execute(self.storage.conn())
    .await?;
    Ok(())
}
```

ethprinter: restrict user's input or make sure the sql statement only affect limited rows

Client Response

client response for yekong: Fixed. fixed in 8be825fa9bf2c56ce684ce4f3d172c8630327036 by restricting user's input and better sql statement.
client response for 0xac: Fixed. fixed in 8be825fa9bf2c56ce684ce4f3d172c8630327036 by restricting user's input and better sql statement.
client response for ethprinter: Fixed. fixed in 8be825fa9bf2c56ce684ce4f3d172c8630327036 by restricting user's input and better sql statement.

ZSU-3:Potential DOS

Category	Severity	Client Response	Contributor
DOS	Low	Acknowledged	Bryce

Code Reference

- code/core/lib/dal/src/proofs_dal.rs#L18-L33
- code/core/lib/dal/src/proofs_dal.rs#L55
- code/core/lib/dal/src/proofs_dal.rs#L74

```
18: sqlx::query!(
19:             r#"
20:             INSERT INTO
21:                 proofs (proof_name, proof, created_at, prover)
22:             VALUES
23:                 ($1, $2, NOW(), $3)
24:             ON CONFLICT (proof_name) DO NOTHING
25:             "#,
26:             proof_name,
27:             proof,
28:             prover,
29:         )
30:         .execute(self.storage.conn())
31:         .await
32:         .unwrap();
33:     }
```

```
55: let record: Vec<u8> = rows.unwrap().proof.unwrap();
```

```
74: .unwrap();
```

Description

Bryce: Insufficient consideration is given to result handling when using sqlx for database operations here. Due to the absence of an error handling mechanism, if an error is returned during a delete operation, it can cause the program to panic and crash, resulting in a DoS attack
The crash log: "thread 'main' panicked... called `Result::unwrap()` on an `Err` value"
Bryce: Insufficient error handling mechanism is in place when using sqlx for database query operations here. If the query returns an empty result set, calling `rows.unwrap()` will cause the program to panic and crash.
The crash log is "thread 'main' panicked... called `Option::unwrap()` on a `None` value"
Bryce: Insufficient handling of result processing when using `sqlx` for database operations here. Due to the lack of error handling mechanism, if the database operation returns an error, it will cause the program to panic and crash, resulting in a denial-of-service attack.
The crash log: "thread 'main' panicked... called `Result::unwrap()` on an `Err` value"

Recommendation

Bryce: It is recommended to use `await?` for error handling. You can refer to the way zksync handles database delete operations in the file `./code/core/lib/dal/src/consensus_dal.rs`

Bryce: Recommendation: If the value of `rows` is `Some`, use `row.proof` to retrieve the result. If the value of `rows` is `None`, you can perform the appropriate error handling logic based on the specific requirements, such as returning a default value or throwing a specific error.
eg.

```
let record: Vec<u8> = match rows {
    Some(row) => row.proof.unwrap(),
    None => {
        // todo
        return Err(MyError::RecordNotFound);
    }
};
```

Bryce: It is recommended to use `await?` for error handling, and when there is no need to return a value, it is advisable not to use `unwrap()`.

Client Response

client response for Bryce: Acknowledged. Acknowledged for that. Current error handling is enough.
client response for Bryce: Acknowledged. Acknowledged for that. Current error handling is enough.
client response for Bryce: Acknowledged. Acknowledged for that. Current error handling is enough.

ZSU-4:No error to handle by `map_err`

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	Yaodao

Code Reference

- code/core/lib/object_store/src/pg.rs#L74

```
74: Ok(()).map_err(|e| ObjectStoreError::from(e))
```

Description

Yaodao: The code provided is a Rust code snippet that uses the map_err function to handle errors. It takes the result of an Ok(()) value and maps it to ObjectStoreError if an error occurs.
However, there is a logical vulnerability in this code. The vulnerability lies in the fact that the **`Ok(())`** value is being used instead of an actual result that could potentially contain an error. This means that the **`map_err`** function will never be executed, as there is no error to handle.

Recommendation

Yaodao: Recommend fixing the logic.

Client Response

client response for Yaodao: Acknowledged. Acknowledged for that. Current error handling is enough.

ZSU-5:Potential `panic` in function `put_raw()`

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	Yaodao

Code Reference

- code/core/lib/object_store/src/pg.rs#L55-75

```
55: async fn put_raw(
56:         &self,
57:         bucket: Bucket,
58:         key: &str,
59:         value: Vec<u8>,
60:     ) -> Result<(), ObjectStoreError> {
61:         let filename = self.filename(bucket, key);
62:         println!("put bucket:{}, key:{}, finename:{}", bucket, key, filename);
63:
64:         let mut storage_processor = self.connection_pool.access_storage().await.unwrap();
65:
66:         let mut transaction = storage_processor.start_transaction().await.unwrap();
67:
68:         transaction
69:             .proof_dal()
70:             .insert_proof(filename, value, "lucy".to_string())
71:             .await;
72:
73:         transaction.commit().await.unwrap();
74:         Ok(()).map_err(|e| ObjectStoreError::from(e))
75:     }
```

Description

Yaodao: The function **`await.unwrap()`** will panic if the value is an **`Err`**, with a panic message provided by the **`Err`**'s value.
However, the return value of the function **`put_raw()`** is declared as **`Result`**.

Recommendation

Yaodao: Recommend confirming this and fixing the logic.

Client Response

client response for Yaodao: Acknowledged. Acknowledged for that. Current error handling is enough.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.