HALBORN

MystenLabs - Denial-of-Service

Security Assessment

Prepared by: **Halborn**

Date of Engagement: **April 14th, 2023 - April 28th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	04/18/2023	Philippe Vogler
0.2	Document Edits	04/28/2023	Erlantz Saenz
0.3	Draft Review	05/01/2023	Gabi Urrutia
0.4	Draft Additions	05/05/2023	Philippe Vogler
0.5	Draft Review	05/09/2023	Erlantz Saenz
1.0	Document Updates	05/12/2023	Erlantz Saenz
1.1	Document Review	05/17/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Philippe Vogler	Halborn	Philippe.Vogler@halborn.com
Erlantz Saenz	Halborn	Erlantz.Saenz@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

MystenLabs engaged Halborn to conduct a Denial-of-Service (DoS) security audit beginning on April 14th, 2023 and ending on April 28th, 2023. The security assessment was scoped to the nodes while performing the attacks through the client, API endpoints and custom Rust implementations.

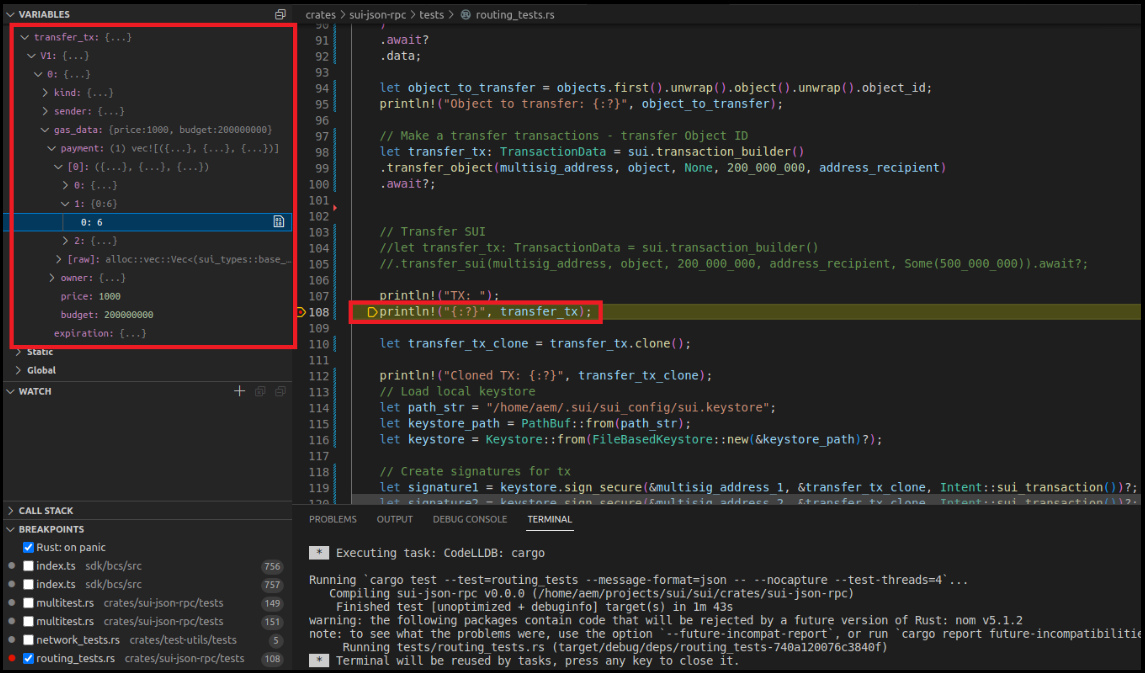
1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to perform a Denial-of-Service audit against nodes in a local environment. The security engineer is an advanced penetration testing expert with knowledge of smart contract security testing and multiple blockchain protocols.

In summary, Halborn did not identify any risk of Denial-of-Service conditions on nodes that may result from custom transactions and signatures, or other attacks performed via such means. Focus was particularly made on creating custom transactions and signatures to cause a node to panic and ultimately crash. For each test, the latency was examined to evaluate any possible slow down when the transaction is being executed. These tests were conducted both on multisig transactions and on transactions with a single signature.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed only manual testing on the scope of this audit. Initial DoS tests were performed via the API endpoints, the CLI and custom transactions written in Rust. The Rust test cases were run through Visual Code’s debugger to edit values on the stack during the test case’s execution, thereby creating custom/malformed transactions or signatures. This was achieved using the Visual Code plugin CodeLLDB and a hexadecimal editor.



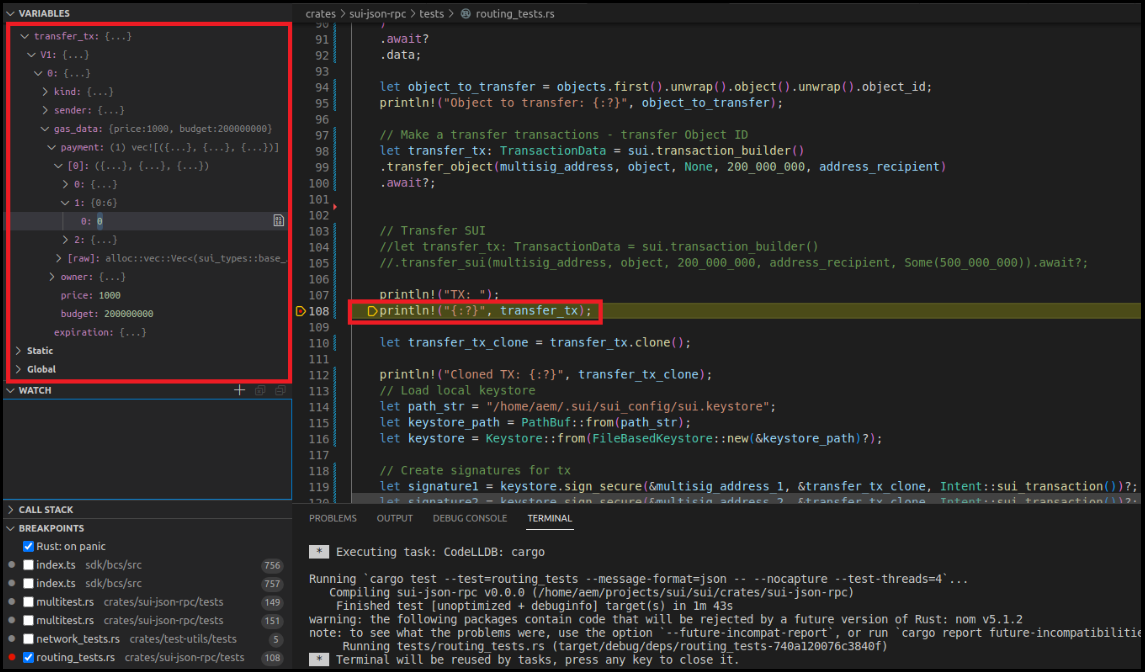
```
VARIABLES
  transfer_tx: {...}
    V1: {...}
      0: {...}
        kind: {...}
        sender: {...}
        gas_data: {price:1000, budget:200000000}
          payment: (1) vec![({...}, {...}, {...})]
            [0]: ({...}, {...}, {...})
              0: {...}
              1: {0:6}
                0: 6
              2: {...}
            [raw]: alloc::vec::Vec<(sui_types::base_…
          owner: {...}
          price: 1000
          budget: 200000000
        expiration: {...}
  Static
  Global
WATCH
CALL STACK
BREAKPOINTS
  ☑ Rust: on panic
  ☐ index.ts  sdk/bcs/src  756
  ☐ index.ts  sdk/bcs/src  757
  ☐ multitest.rs  crates/sui-json-rpc/tests  149
  ☐ multitest.rs  crates/sui-json-rpc/tests  151
  ☐ network_tests.rs  crates/test-utils/tests  5
  ☑ routing_tests.rs  crates/sui-json-rpc/tests  108
```

```rust
crates > sui-json-rpc > tests > routing_tests.rs
    .await?
    .data;

    let object_to_transfer = objects.first().unwrap().object().unwrap().object_id;
    println!("Object to transfer: {:?}", object_to_transfer);

    // Make a transfer transactions - transfer Object ID
    let transfer_tx: TransactionData = sui.transaction_builder()
    .transfer_object(multisig_address, object, None, 200_000_000, address_recipient)
    .await?;


    // Transfer SUI
    //let transfer_tx: TransactionData = sui.transaction_builder()
    //.transfer_sui(multisig_address, object, 200_000_000, address_recipient, Some(500_000_000)).await?;

    println!("TX: ");
    println!("{:?}", transfer_tx);

    let transfer_tx_clone = transfer_tx.clone();

    println!("Cloned TX: {:?}", transfer_tx_clone);
    // Load local keystore
    let path_str = "/home/aem/.sui/sui_config/sui.keystore";
    let keystore_path = PathBuf::from(path_str);
    let keystore = Keystore::from(FileBasedKeystore::new(&keystore_path)?);

    // Create signatures for tx
    let signature1 = keystore.sign_secure(&multisig_address_1, &transfer_tx_clone, Intent::sui_transaction())?;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
 *  Executing task: CodeLLDB: cargo

Running `cargo test --test=routing_tests --message-format=json -- --nocapture --test-threads=4`...
   Compiling sui-json-rpc v0.0.0 (/home/aem/projects/sui/sui/crates/sui-json-rpc)
    Finished test [unoptimized + debuginfo] target(s) in 1m 43s
warning: the following packages contain code that will be rejected by a future version of Rust: nom v5.1.2
note: to see what the problems were, use the option `--future-incompat-report`, or run `cargo report future-incompatibilitie
     Running tests/routing_tests.rs (target/debug/deps/routing_tests-740a120076c3840f)
 *  Terminal will be reused by tasks, press any key to close it.
```

Figure 1: Example of replacing the object version in a custom transaction during runtime with CodeLLDB - Original version is 6.



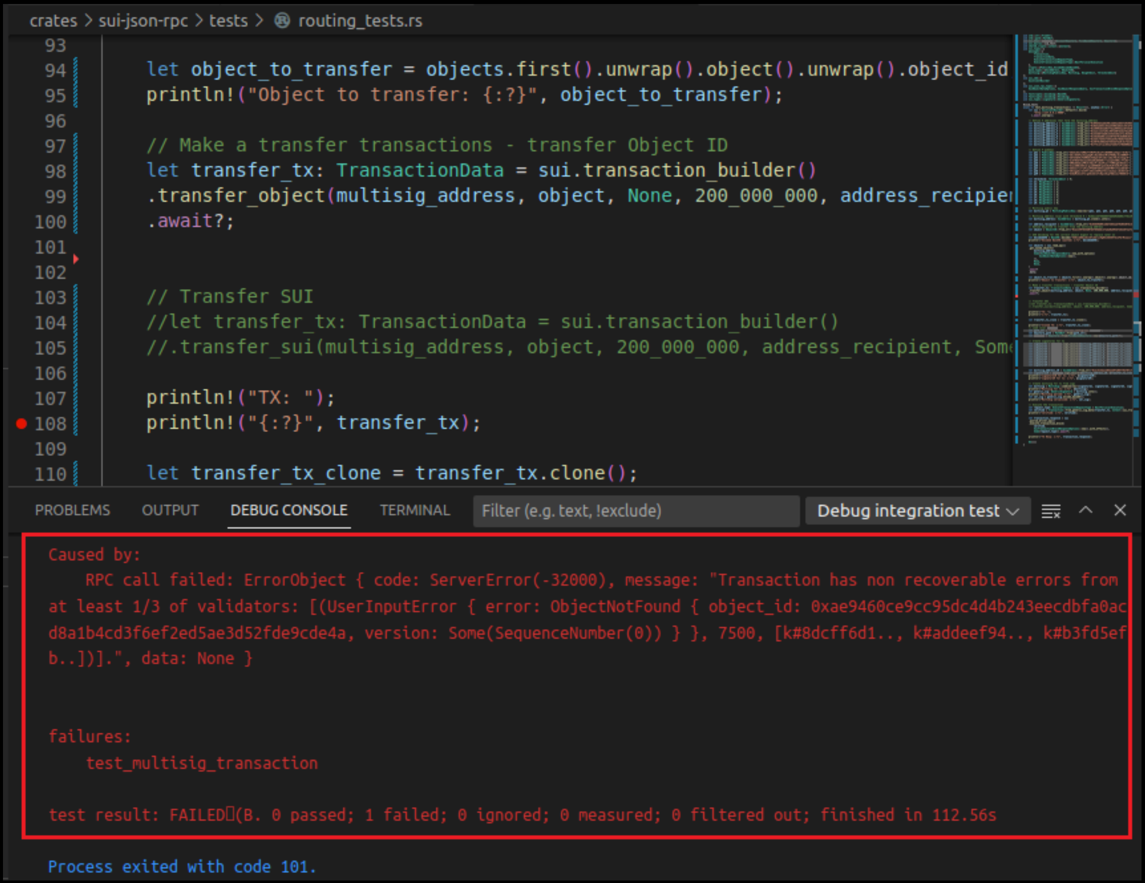
```
VARIABLES
  transfer_tx: {...}
    V1: {...}
      0: {...}
        kind: {...}
        sender: {...}
        gas_data: {price:1000, budget:200000000}
          payment: (1) vec![({...}, {...}, {...})]
            [0]: ({...}, {...}, {...})
              0: {...}
              1: {0:6}
                0: 0
              2: {...}
            [raw]: alloc::vec::Vec<(sui_types::base_…
          owner: {...}
          price: 1000
          budget: 200000000
        expiration: {...}
  Static
  Global
WATCH
CALL STACK
BREAKPOINTS
  ☑ Rust: on panic
  ☐ index.ts  sdk/bcs/src  756
  ☐ index.ts  sdk/bcs/src  757
  ☐ multitest.rs  crates/sui-json-rpc/tests  149
  ☐ multitest.rs  crates/sui-json-rpc/tests  151
  ☐ network_tests.rs  crates/test-utils/tests  5
  ☑ routing_tests.rs  crates/sui-json-rpc/tests  108
```

```rust
crates > sui-json-rpc > tests > routing_tests.rs
    .await?
    .data;

    let object_to_transfer = objects.first().unwrap().object().unwrap().object_id;
    println!("Object to transfer: {:?}", object_to_transfer);

    // Make a transfer transactions - transfer Object ID
    let transfer_tx: TransactionData = sui.transaction_builder()
    .transfer_object(multisig_address, object, None, 200_000_000, address_recipient)
    .await?;


    // Transfer SUI
    //let transfer_tx: TransactionData = sui.transaction_builder()
    //.transfer_sui(multisig_address, object, 200_000_000, address_recipient, Some(500_000_000)).await?;

    println!("TX: ");
    println!("{:?}", transfer_tx);

    let transfer_tx_clone = transfer_tx.clone();

    println!("Cloned TX: {:?}", transfer_tx_clone);
    // Load local keystore
    let path_str = "/home/aem/.sui/sui_config/sui.keystore";
    let keystore_path = PathBuf::from(path_str);
    let keystore = Keystore::from(FileBasedKeystore::new(&keystore_path)?);

    // Create signatures for tx
    let signature1 = keystore.sign_secure(&multisig_address_1, &transfer_tx_clone, Intent::sui_transaction())?;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
 *  Executing task: CodeLLDB: cargo

Running `cargo test --test=routing_tests --message-format=json -- --nocapture --test-threads=4`...
   Compiling sui-json-rpc v0.0.0 (/home/aem/projects/sui/sui/crates/sui-json-rpc)
    Finished test [unoptimized + debuginfo] target(s) in 1m 43s
warning: the following packages contain code that will be rejected by a future version of Rust: nom v5.1.2
note: to see what the problems were, use the option `--future-incompat-report`, or run `cargo report future-incompatibilitie
     Running tests/routing_tests.rs (target/debug/deps/routing_tests-740a120076c3840f)
 *  Terminal will be reused by tasks, press any key to close it.
```

Figure 2: Example of replacing the object version custom transaction during runtime with CodeLLDB - Value replaced with 0.


crates > sui-json-rpc > tests > routing_tests.rs
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, !exclude)
Debug integration test

```
    let object_to_transfer = objects.first().unwrap().object().unwrap().object_id
    println!("Object to transfer: {:?}", object_to_transfer);

    // Make a transfer transactions - transfer Object ID
    let transfer_tx: TransactionData = sui.transaction_builder()
    .transfer_object(multisig_address, object, None, 200_000_000, address_recipie
    .await?;


    // Transfer SUI
    //let transfer_tx: TransactionData = sui.transaction_builder()
    //.transfer_sui(multisig_address, object, 200_000_000, address_recipient, Som

    println!("TX: ");
    println!("{:?}", transfer_tx);

    let transfer_tx_clone = transfer_tx.clone();
```

```
Caused by:
    RPC call failed: ErrorObject { code: ServerError(-32000), message: "Transaction has non recoverable errors from
at least 1/3 of validators: [(UserInputError { error: ObjectNotFound { object_id: 0xae9460ce9cc95dc4d4b243eecdbfa0ac
d8a1b4cd3f6ef2ed5ae3d52fde9cde4a, version: Some(SequenceNumber(0)) } }, 7500, [k#8dcff6d1.., k#addeef94.., k#b3fd5ef
b..])].", data: None }


failures:
    test_multisig_transaction

test result: FAILED(B. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 112.56s

Process exited with code 101.
```

Figure 3: Example of replacing the object version custom transaction during runtime with CodeLLDB - Failed transaction.

Two Rust test cases have been written for this engagement, respectively to send transactions from a multisig address, or single address. A multisig transaction requires more than one private key to authorize it. The latter implementation can be used to transfer SUI or objects with a single signature, while the former handles the multisig transaction. They can be found in the section CUSTOM RUST IMPLEMENTATIONS later in this report.

The following test cases shared by MystenLabs prior to the beginning of the engagement were also considered:
- Send valid signatures, but the object version is not current (which means sig verification will succeed)
- Send valid multi-sig 10-out-10, but the object version is not current (which means sig verification will succeed) - worst-case scenario is all signatures are ECDSA r1 (which is the slowest in verification)
- Send invalid signatures, but correct tx
- Send invalid multi-sig 10-out-10 where just the final sig fails - worst-case scenario is all signatures are ECDSA r1 (which is the slowest in verification)

Further information about the tests conducted via the CLI, API endpoints and the Rust implementations may be found in the last section of this report - TESTS CONDUCTED.

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

Denial-of-Service (DoS) attacks performed against the nodes.
Tests were conducted via:
- CLI
- API endpoints
- Custom transactions written in Rust

Documentation provided:
- https://docs.sui.io/

1.5 CAVEATS

During the first week of the engagement, the test cases provided by MystenLabs were attempted via the CLI and the API endpoints. MystenLabs later mentioned that both the CLI and API endpoints are hardened against malicious user inputs, and instead, custom transactions should be written from scratch. As such, during the second week, in-depth testing of transactions and signatures was conducted via custom Rust implementations to test transactions having a single signature, and multisig transactions.

2. CUSTOM RUST IMPLEMENTATIONS

2.1 SINGLE SIGNATURE TRANSACTIONS

Listing 1: test_singlesig_transaction.rs

```rust
// Halborn Single Transaction Tests
use std::path::PathBuf;
use sui_json_rpc_types::{
    SuiObjectDataOptions, SuiObjectResponseQuery,
↳ SuiTransactionBlockResponseOptions
};
use sui_keys::keystore::{AccountKeystore, FileBasedKeystore,
↳ Keystore};
use sui_macros::sim_test;
use shared_crypto::intent::{Intent};

use std::str::FromStr;
use sui_sdk::{
    SuiClientBuilder,
    types::{
        base_types::{SuiAddress,ObjectID},
        messages::Transaction
    }
};
use sui_types::messages::{
    ExecuteTransactionRequestType,
    ExecuteTransactionRequestType::WaitForLocalExecution,
    TransactionData
};

#[sim_test]
async fn test_singlesig_transaction() -> Result<(), anyhow::Error>
↳ {

    // Set with the local lab
    let sui = SuiClientBuilder::default().build(
        "http://127.0.0.1:9000",
      ).await.unwrap();

    // Use the local addresses & object
    let address_signer = SuiAddress::from_str("0x3b4cfd5527365ad39a70b82fbde93189ec4c94d5a37f69a7f294809c32a0642a")?;
    let object = ObjectID::from_str("0x384f4af89a66ff4b660352c8575b254c4097f2c2967cc18c3d10e9c5eed71944")?;
    let address_recipient = SuiAddress::from_str("0x84659d09ce8d73852abf9e052076c33092745f60c45f3a9406825128170c28c2")?;

    // Two options below: transfer SUI or transfer object
    // Transfer SUI
    //let transfer_tx: TransactionData = sui.transaction_builder()
    //.transfer_sui(address_signer, object, 200_000_000, address_recipient, Some(500_000_000)).await?;

    // transfer object
    let transfer_tx: TransactionData = sui.transaction_builder()
        .transfer_object(address_signer, object, None, 200_000_000, address_recipient)
        .await?;

    // Set a breakpoint on the println! statement to modify transfer_tx
    println!("TX: {:?}", transfer_tx);

    // Local keystore
    let path_str = "/home/aem/.sui/sui_config/sui.keystore";
    let keystore_path = PathBuf::from(path_str);
    let keystore = Keystore::from(FileBasedKeystore::new(&keystore_path)?);

    // Sign TX
    let signature = keystore.sign_secure(&address_signer, &transfer_tx, Intent::sui_transaction())?;
    println!("Signature: {:?}", signature);

    // Execute the transaction
    let request_type: ExecuteTransactionRequestType = WaitForLocalExecution;
    let transaction_response = sui
    .quorum_driver_api()
    .execute_transaction_block(
        Transaction::from_data(transfer_tx, Intent::sui_transaction(), vec![signature]).verify().expect("transaction"),
        SuiTransactionBlockResponseOptions::new().with_effects(),
        Some(request_type)).await?;

    println!("TX Response {:?}", transaction_response);

    Ok(())
}
```

2.2 MULTI SIGNATURE TRANSACTIONS

Listing 2: test_multisig_transaction.rs

```rust
// Halborn Multi Transaction Tests

use std::str::FromStr;
use std::path::PathBuf;
use sui_keys::keystore::{AccountKeystore, FileBasedKeystore,
↳ Keystore};
use sui_macros::sim_test;
use shared_crypto::intent::{Intent};
use sui_types::{
    messages::{
        Transaction,
        TransactionData,
        ExecuteTransactionRequestType,
        ExecuteTransactionRequestType::WaitForLocalExecution
    },
    crypto::{PublicKey,EncodeDecodeBase64},
    base_types::{SuiAddress, ObjectID},
    multisig::{MultiSigPublicKey, MultiSig, WeightUnit,
↳ ThresholdUnit}
};
use sui_sdk::{
    SuiClientBuilder
};
use sui_json_rpc_types::{
    SuiObjectDataOptions, SuiObjectResponseQuery,
↳ SuiTransactionBlockResponseOptions
};
use fastcrypto::encoding::Base58;
use fastcrypto::encoding::Encoding;
use sui_types::signature::GenericSignature;

#[sim_test]
async fn test_multisig_transaction() -> Result<(), anyhow::Error>
↳ {
    let sui = SuiClientBuilder::default().build(
        "http://127.0.0.1:9000",
      ).await.unwrap();

    // Define 9 addresses
    let multisig_address_1 = SuiAddress::from_str("0x09d5d3ad8290c2d85a2863fe9298427c7241542c0c1c645a06cc0b7e1363baae")?;
    let multisig_address_2 = SuiAddress::from_str("0x0d3ab652c63ae53081d8e3c45cbfbf7eda3cd92b7154169514fb6c0b00ba97b9")?;
    let multisig_address_3 = SuiAddress::from_str("0x2999283eb02ffdaa9955ea18f24e9a7756695c578d1db841dedd0138fb306b21")?;
    let multisig_address_4 = SuiAddress::from_str("0x2a2c1227a92c987e607beba76aa9574116dbf2659144d554810728f4d2ded24e")?;
    let multisig_address_5 = SuiAddress::from_str("0x3338873dd01335b328b137ac0e5d500784ceba6e1fc1a4a6d0dcc1b646ae4b6f")?;
    let multisig_address_6 = SuiAddress::from_str("0x38e8bd08ca2e30f5d7916b060182520794368198 8eb3d767abbb9f5502c533c2")?;
    let multisig_address_7 = SuiAddress::from_str("0x3b4cfd5527365ad39a70b82fbde93189ec4c94d5a37f69a7f294809c32a0642a")?;
    let multisig_address_8 = SuiAddress::from_str("0x50f91608e5d7563bbfba0c617d22832538b12c8848a3a0c2c6c319c0ec3c5e93")?;
    let multisig_address_9 = SuiAddress::from_str("0x774ce7cac642afeb62f1f0b600d2860d750f2040026546b016d522c722b7a839")?;

    // Define 9 pubkeys
    let pk1 = PublicKey::from_str("AKk5x6hxTbMWwYT3MkPh10xwFt1WGRW0syVZqnr+8v0o").unwrap();
    let pk2 = PublicKey::from_str("AOclOS6Fyx5wyp6gMlzRanK83ktMveY9GNp/5xtKNMXF").unwrap();
    let pk3 = PublicKey::from_str("AOfS4QV9LFZHMZ6sV5DpAlGFcCKJr1OmcFELVztZagl9").unwrap();
    let pk4 = PublicKey::from_str("AL8f0kdTGXxew2E4wd0uQ8UD0v++ooJp3I5WKL/YTL8e").unwrap();
    let pk5 = PublicKey::from_str("AMDo8UZdxyPMJ4lVQysP/LOuSPialr74MK1gUvRAIXvc").unwrap();
    let pk6 = PublicKey::from_str("AJyzGWB1X85xb/gxwQH0WdRlpeknDw9yRtZ2DzPcy3O1").unwrap();
    let pk7 = PublicKey::from_str("AEhI2v+it+aPzwjPFd607bDVInydADfMRTv7nwnUUtQ7").unwrap();
```

```rust
    let pk8 = PublicKey::from_str("AHn2gq8xrZn5Ff/ZO21I66Pq2T41zvmykaa/LK1EZt1G").unwrap();
    let pk9 = PublicKey::from_str("AIUaMxS1s3fn/gh81ZAv2fv0psIEjpfOQ1xeiT0O8KvX").unwrap();

    let threshold: ThresholdUnit = 9;
    let w1: WeightUnit = 1;
    let w2: WeightUnit = 2;
    let w3: WeightUnit = 3;
    let w4: WeightUnit = 4;
    let w5: WeightUnit = 5;
    let w6: WeightUnit = 6;
    let w7: WeightUnit = 7;
    let w8: WeightUnit = 8;
    let w9: WeightUnit = 9;

    // Multisig public key
    let multisig_pk = MultiSigPublicKey::new(vec![pk1, pk2, pk3, pk4, pk5, pk6, pk7, pk8, pk9], vec![w1, w2, w3, w4, w5, w6, w7, w8, w9], threshold).unwrap();

    // Local multisig address with threshold 9 = 0x6b1c2bf58b684fbb5b926d06c7f61194d338e253988d5bb37b078ae91b276b76
    let multisig_address: SuiAddress = multisig_pk.clone().into();

    let address_recipient = SuiAddress::from_str("0x84659d09ce8d73852abf9e052076c33092745f60c45f3a9406825128170c28c2")?;
    // Define ObjectID to transfer from the multisig address
    let object = ObjectID::from_str("0xc44d88a04569eeb02f59143dd6984e3d3620839bfd6a2bea03ae1d041dc1f629")?;

    /* Used to calculate the digest of an object ID's version - replace the value with the hex editor
    let decodedb58 = Base58::decode("4XDJidD8jJGsvG7LXkrLz9gNhxSU4EV7Ai2FmrMzoass");
    println!("Decoded Base58 (wanted) {:?}", decodedb58);
    */

    // Two options below: transfer SUI or transfer object
    // transfer object
    let transfer_tx: TransactionData = sui.transaction_builder()
    .transfer_object(multisig_address, object, None, 200_000_000, address_recipient)
    .await?;

    // Transfer SUI
    //let transfer_tx: TransactionData = sui.transaction_builder()
    //.transfer_sui(multisig_address, object, 200_000_000, address_recipient, Some(500_000_000)).await?;

    println!("TX: {:?}", transfer_tx);

    let transfer_tx_clone = transfer_tx.clone();

    // Load local keystore
    let path_str = "/home/aem/.sui/sui_config/sui.keystore";
    let keystore_path = PathBuf::from(path_str);
    let keystore = Keystore::from(FileBasedKeystore::new(&keystore_path)?);

    // Create signatures for tx
    let signature1 = keystore.sign_secure(&multisig_address_1, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature2 = keystore.sign_secure(&multisig_address_2, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature3 = keystore.sign_secure(&multisig_address_3, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature4 = keystore.sign_secure(&multisig_address_4, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature5 = keystore.sign_secure(&multisig_address_5, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature6 = keystore.sign_secure(&multisig_address_6, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature7 = keystore.sign_secure(&multisig_address_7, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature8 = keystore.sign_secure(&multisig_address_8, &transfer_tx_clone, Intent::sui_transaction())?;
    let signature9 = keystore.sign_secure(&multisig_address_9, &transfer_tx_clone, Intent::sui_transaction())?;

    // Create multisig for tx from sigs
    let multisig = MultiSig::combine(vec![signature1, signature2, signature3, signature4, signature5, signature6, signature7, signature8, signature9], multisig_pk).unwrap();
    println!("Multisig for tx: {:?}", &multisig);
    let generic_sig: GenericSignature = multisig.into();
    println!("MultiSig parsed: {:?}", generic_sig);
    let ser_sig = generic_sig.encode_base64();
    println!("MultiSig serialized: {:?}", ser_sig);

    // Execute the transaction
    let request_type: ExecuteTransactionRequestType = WaitForLocalExecution;
    let verified = Transaction::from_generic_sig_data(transfer_tx, Intent::sui_transaction(), vec![generic_sig]).verify()?;
    println!("verified: {:?}", verified);

    let transaction_response = sui
    .quorum_driver_api()
    .execute_transaction_block(
        verified,
        SuiTransactionBlockResponseOptions::new().with_effects(),
        Some(request_type)).await?;

    println!("TX Resp: {:?}", transaction_response);

    Ok(())
}
```

3. TESTS CONDUCTED

3.1 API

- Endpoints attacked: transfer_sui and transfer_object
- Parameters tested: signer, sui_object_id, gas_budget, recipient, amount, object_id, gas
- Test cases conducted:
  - Fuzz input parameters with max values (e.g. 65535 for 2 bytes, 18446744073709551615 for u64), null bytes, empty values, improper types.
  - Duplicate JSON parameters/values to test for overflows.
  - Normalization attacks to detect any improper parsing/crash.
  - Send large HTTP requests.
  - Send invalid signatures/tx bytes by modifying the base64 encoding.

3.2 CLI

- Forge malformed multisig address.
- Combine more than 10 signatures.
- Combine signatures when one of them is invalid.

3.3 Rust Implementations

Endpoints attacked: transfer_sui and transfer_object

- Reuse old object version of an already transferred object (single & multisig).
  - Parameters: object version and object digest

  - Result: Cannot be reused since the object ID does not exist anymore. Transaction has non-recoverable errors from at least 1/3 of validators: [...] { error: ObjectNotFound

- Reuse old object version of the transfer object (single & multisig).
  - Parameters: object version
  - Result: Transaction has non-recoverable errors from at least 1/3 of validators: [...] { error: ObjectNotFound

- Reuse old object version when paying SUI to another address (single & multisig).

  a. Two test cases were considered - if objectdigest of the object version is incorrect:
     the object version in transfer_tx was modified **before** being copied with transfer_tx_clone = transfer_tx.clone(). This means all subsequent tx signing is done on the modified value.
     Result: Transaction has non-recoverable errors from at least 1/3 of validators: [(UserInputError { error: ObjectNotFound { object_id: 0x044ebf3bce8cddb661a521fefde385a455a02af85581f431bd76463afd10
     , version: Some(SequenceNumber(1))} }
     the object version in transfer_tx was modified **after** being copied with transfer_tx_clone = transfer_tx.clone(). This means all subsequent tx signing is done on the initial value.
     Result: Error: Signature is not valid: Invalid signature for pk="AKk5x6hxTbMWwYT3MkPh10xwFt1WGRW0syVZqnr+8v0o"

  b. If the function is transfer_object instead of transfer_sui, the errors remain the same as described in a.

- Test SUI transfers with varying object versions and object digests (single & multisig).
  - Parameters: object version and object digest
  - Result: The transaction works only if the object version and digest are correct. Found no way to abuse it for a DoS attack.

- Test object transfers with varying versions and digests (single & multisig).
  - Parameters: object version and object digest
  - Result: The transaction works only if the object version and digest are correct. Found no way to abuse it for a DoS attack.

- Reuse same object ID for both the gas and SUI transfer (single & multisig).
  - Parameters: `object_id` and `gas`
  - Result: Denied. Each object needs to be unique.
- Send invalid signatures but correct tx.
  - Parameters: each `signature[i]` of the multisig, where 0 <= i <= 8.
  - Result: Each signature is properly checked. Changing one byte within the multisig ultimately results in an error `Error: Signature is not valid: Invalid signature for pk="public_key"`
- Change all the bytes of each signature to 0xFF to detect overflows.
  - Parameters: each `signature[i]` of the multisig, where 0 <= i <= 8.
  - Result: Each signature is properly checked. Changing one byte within the multisig ultimately results in an error `Error: Signature is not valid: Invalid signature for pk="public_key"`
- Create 3 multi addresses with each of the three allowed cryptographic signature algorithms and compare the transaction processing time when the last signature's last byte is changed.
  - Result: No visible processing time difference.
- Change one byte of the last signature of a multisig composed of 10 sigs.
  - Parameter: `signature[i]` where i is the last signature ID (0 <= i <= 8)
  - Result: Each signature is properly checked. Changing one byte within the multisig ultimately results in an error `Error: Signature is not valid: Invalid signature for pk="public_key"`
- Change one byte in each signature of a multisig composed of 10 sigs.
  - Parameters: each `signature[i]` of the multisig, where 0 <= i <= 8.
  - Result: Each signature is properly checked. Changing one byte within the multisig ultimately results in an error `Error: Signature is not valid: Invalid signature for pk="public_key"`

- Null out or max value in each signature of a multisig composed of 10 sigs.
  - Parameters: each signature[i] of the multisig, where 0 <= i <= 8.
  - Result: Each signature is properly checked. Changing one byte within the multisig ultimately results in an error Error: Signature is not valid: Invalid signature for pk="public_key"
- Null out or max value the last signature of a multisig composed of 10 sigs.
  - Parameter: signature[i] where i is the last signature ID (0 <= i <= 8)
  - Result: Each signature is properly checked. Changing one byte within the multisig ultimately results in an error Error: Signature is not valid: Invalid signature for pk="public_key"
- Add more than 10 signatures to a transaction.
  - Result: InvalidSignature { error: "Invalid number of signatures" }
- Send u64 max in fields
  - Parameters: gas price, gas budget, amount, threshold, weight.
  - Result: Thresholds are checked for each field. For example, the gas price cannot exceed 100_000 Mist.
- Send u16 max in several fields within the TransactionEnveloppe:
  - Parameters: each index of the payment structure and of the kind structure, object version, threshold
  - Result: Thresholds are checked for each field.
- Replace bytes to 0xFF in parameters to check for crashes or improper parsing.
  - Parameters: each decimal value within input, object version, transactionData version, parameters in commands, array gas_data , owner, price, budget, threshold.
  - Result: Various error messages are returned based on the parameter modified. Found no possible crash of the node for any of the tested parameters.

- Replace some bytes within the TransactionEnveloppe to 0xFF to check for crashes from an improperly formatted tx.
    - Result: Memory error SIGSEGV is returned to the debugger but observed no node crash.
- Tamper with the Pure decimal values from the transaction.
    - Parameters: recipient and amount
    - Result: Transaction fails if bytes related to the recipient are changed, otherwise the amount sent changes.
- Fuzz input parameters with max values (e.g. 65535 for 2 bytes, 18446744073709551615 for u64), null bytes, empty values and improper types.
    - Parameters tested: sender, amount, initialSharedVersion, messageVersion, owner, price, budget, object version.
    - Result: Observed node crash on gas price. Fixed as of 8dfd3da78825486829a8d8684d4c374597b76b81. Various error messages are returned based on the parameter modified.

THANK YOU FOR CHOOSING

// HALBORN