SHERLOCK

Security Review For RuneMine

Private Contest Prepared For:	RuneMine
Lead Security Expert:	cergyk
Date Audited:	November 19 - November 28, 2024

Introduction

MineLabs (formerly RuneMine) build tooling for the Runes fungible token ecosystem on Bitcoin L1. Our main focus and product at the moment is a bridge bringing Runes tokens to other chain or helping existing token bridge to Runes on L1.

Scope

Repository: runemine/bridge

Branch: ref

Audited Commit: 76ca43cb0a519c81b0045efede9b8e584152e4da

Final Commit: 4a24aac64756588da1a307931bcb50f8d1f01ac7

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
7	3

Issues Not Fixed or Acknowledged

High	Medium
0	0

Security experts who found valid issues

cergyk	0xeix	dod4ufn
KupiaSec	Atharv	TessKimy

Issue H-1: Some btc incoming transfers can be skipped due to panic induced by invalid evm address

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/3

Found by

cergyk

Summary

The bridge uses polling of rune `utxos` to `ms` address (2500 at a time) to determine which transfers must be executed on other chains (solana or bitlayer evm). When parsing an evm address from the runestone field "43", if the data is too short (e.g only one uint128), an unhandled panic will be triggered (`runtimeerror:sliceboundsoutofrange[:20]withcapacity16`).

This will not halt the bridge program because errors of the go routine are still recovered, but the bridge will attempt to scan the transfer and fail repeatedly until it can be skipped because more than 2500 `utxos` are available.

In that case some other transfers may be skipped alongside the invalid one with high probability.

Root Cause

In chain_btc.go#L37-L53:

```
  if ta, ok := tx.GetJ("fields")["43"]; ok {
      bs := []byte{}
      for i, n := range ta.([]interface{}) {
          bss := StringToBi(n.(string)).Bytes()
          slices.Reverse(bss)
          bs = append(bs, bss...)
          for len(bs) < (i+1)*16 {
              bs = append(bs, 0) // when number starts with 0 we will be missing
 ↪  bytes
          }
      }
      if targetNetwork == NetworkSolana {
          targetAddress = BytesToBase58(bs)
      } else if targetNetwork == NetworkBitlayer {
          // TODO generalize condition for all evm targets
```

```
                  //@audit can be only 16 bytes long if only 1 uint128 is passed
 ↪  into the runestone field 43
>>          targetAddress = hexutil.Encode(bs[:20])
        }
    }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. A malicious actor sends some very small amount of any rune (even some rune without value which is not whitelisted by the system because this is not checked during the scan), and adds the message field "43" with only 1 uint128.

Impact

1/ Processing of incoming transfers on btc chain is blocked until the window of the indexer query is reached (2500) 2/ The bridge scanner will skip some utxos with high probability during high usage periods (due to the fact that the processing halted until reaching the limit of 2500 utxos).

> Indeed, unless the limit of 2500 is crossed by exactly 1 utxo, some utxos will be skipped (because only latest 2500 are processed)

PoC

Add the following test to `bridge_test.go`:

```
func TestInvalidEvmAddressDecode(t *testing.T) {
    ta := [1]string{}
    ta[0] = "43962546600517487046958341907937552909"
    bs := []byte{}
    for i, n := range ta {
        bss := StringToBi(n).Bytes()
        slices.Reverse(bss)
        bs = append(bs, bss...)
        for len(bs) < (i+1)*16 {
            bs = append(bs, 0) // when number starts with 0 we will be missing bytes
        }
    }
    // TODO generalize condition for all evm targets
    hexutil.Encode(bs[:20])
}
```

```
 go test ./bridge -run "TestInvalidEvmAddressDecode" -test.v
=== RUN   TestEvmAddressDecode
--- FAIL: TestEvmAddressDecode (0.00s)
panic: runtime error: slice bounds out of range [:20] with capacity 16 [recovered]
    panic: runtime error: slice bounds out of range [:20] with capacity 16

goroutine 20 [running]:
testing.tRunner.func1.2({0xb618e0, 0xc0001486d8})
    /usr/lib/go/src/testing/testing.go:1632 +0x230
testing.tRunner.func1()
    /usr/lib/go/src/testing/testing.go:1635 +0x35e
panic({0xb618e0?, 0xc0001486d8?})
    /usr/lib/go/src/runtime/panic.go:785 +0x132
runemine-bridge/bridge.TestEvmAddressDecode(0xc000167380?)
    /home/xxx/sherlock/contests/2024-11-runemine/bridge/bridge/bridge_test.go:496
     ↪  +0x20b
testing.tRunner(0xc000167380, 0xcb8be0)
    /usr/lib/go/src/testing/testing.go:1690 +0xf4
created by testing.(*T).Run in goroutine 1
    /usr/lib/go/src/testing/testing.go:1743 +0x390
FAIL    runemine-bridge/bridge0.015s
FAIL
```

Mitigation

In the case the target is EVM, please consider padding the bytes array `bs` to be at least 20 bytes long.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/runemine/bridge/pull/20

Issue H-2: Malicious user can trick bridge into spending rune utxos if associated value is bigger than `minimumDustThreshold`

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/4

Found by

cergyk

Summary

During outgoing rune transfers on the btc chain, the bridge logic checks which utxos can be used to spend value needed for the rune transfers (minimum dust `564` to associate with destination output and "change" output, and tx fee). It skips the `utxos` having less than `minimumDustThreshold` because those are associated with incoming rune transfers. Unfortunately a malicious user can trick the bridge into using incoming rune transfers by associating a value of `10001` with a rune output.

Since the constructed transaction will not have a runestone output, it means that the runes associated with that output will simply be lost and unclaimable by anyone, causing consequent losses for the bridge.

Root Cause

util_btc.go#L97-L99:

```
    for _, utxo := range utxos {
        value := StringToInt(utxo.Get("value"))

        //@audit if value > minimumDustThreshold the utxo is a candidate to be
 ↪  spent!
>>      if value <= minimumDustThreshold {
>>          continue // don't spend what is most likely to be a rune utxo
>>      }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Malicious user makes a transaction in which the output directed to bridge address and which will be the target of the rune transfer, has an associated value of `10001`.

Impact

The bridge will use that output at some point to pay for transaction fees, and will lose associated runes. The attacker does not lose anything since he is able to claim tokens on destination chain and roundtrip back on btc taking somebody else rune balance.

As a result the bridge is insolvent in the targeted rune token.

PoC

No response

Mitigation

Consider checking if the candidate utxo is associated with a rune explicitely

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/12

Issue H-3: Sol scan may be DoSed and locks may be skipped due to unhandled panic

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/8

Found by

cergyk, dod4ufn

Summary

The bridge uses polling of solana signatures to the address of the bridge `ms`. Inside these signature objects, individual instructions are parsed and parameters for transfers are extracted. One of these parameters is the native fee to be paid to the bridge. Unfortunately if the fee is insufficient a panic is triggered in the bridge code.

Due to the recovery of the go routine, the bridge will attempt to iterate on the failed signature until reaching window limit of `100` (polling window of the indexer), at which point some signatures will be skipped during high usage.

Root Cause

chain_sol.go#L66-L104:

```
func (b *Bridge) solScanBurn(id, signer string, blockTime time.Time, decoder
 ↪  *SolanaDecoder, mint string) *Transfer {
    _ = decoder.String()
    amount := IntToBi(int64(decoder.Uint64()))
    fee := decoder.Uint64()
    targetNetwork := decoder.Uint8()
    targetAddress := decoder.String()
    if fee < 50000000 {
>>      panic(fmt.Sprintf("sol fee paid to small to cover transaction cost: %d",
 ↪  fee))
    }
    return &Transfer{
        ID:      id,
        Target:  int(targetNetwork),
        From:    signer,
        To:      targetAddress,
        Rune:    mint,
        Amount:  amount.String(),
        Created: blockTime,
    }
}
```

```
// parses a lock instruction details into a transfer
func (b *Bridge) solScanLock(id, signer string, blockTime time.Time, decoder
 ↪  *SolanaDecoder, mint string) *Transfer {
    amount := IntToBi(int64(decoder.Uint64()))
    fee := decoder.Uint64()
    targetNetwork := decoder.Uint8()
    targetAddress := decoder.String()
    if fee < 50000000 {
>>      panic(fmt.Sprintf("sol fee paid to small to cover transaction cost: %d",
 ↪  fee))
    }
    return &Transfer{
        ID:      id,
        Target:  int(targetNetwork),
        From:    signer,
        To:      targetAddress,
        Rune:    mint,
        Amount:  amount.String(),
        Created: blockTime,
    }
}
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Malicious user locks small amount of tokens to solana bridge program and provides too small of a fee to be processed by the bridge

Impact

Some user events will be skipped with high probability, causing loss of locked funds for users

PoC

No response

Mitigation

Please consider recovering from these errors gracefully by saving the transfer with an `error` or `ignored` status

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/21

Issue H-4: There is no mechanism in the EVM Bridge to cover gas fees

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/20

Found by

0xeix, KupiaSec

Summary

There is a mechanism for covering gas fees in the Solana Bridge, but none exists in the EVM Bridge, leading to potential fund losses for the protocol.

Root Cause

As you can see in the burn() function of the Solana Bridge program, certain fees are incurred.

```
    pub fn burn(
        ctx: Context<Burn>,
        name: String,
        amount: u64,
@>      fee: u64,
        target_network: u8,
        target_address: String,
    ) -> Result<()> {
        emit!(EventBurn {
            from: *ctx.accounts.signer.to_account_info().key,
            mint: *ctx.accounts.mint.to_account_info().key,
            name: name.clone(),
            amount,
@>          fee,
            target_network,
            target_address,
        });
        if fee > 0 {
            anchor_lang::system_program::transfer(
                CpiContext::new(
                    ctx.accounts.system_program.to_account_info(),
                    anchor_lang::system_program::Transfer {
                        from: ctx.accounts.signer.to_account_info(),
                        to: ctx.accounts.bridge_signer.to_account_info(),
                    },
                ),
@>              fee,
            )?;
        }
            [... ...]
    }
```

This fee covers the gas cost when the protocol initiates a transaction to mint the corresponding tokens on the target network and should not be less than 50,000,000 lamports (0.05 SOL, approximately $10).

https://github.com/runemine/bridge/tree/76ca43cb0a519c81b0045efede9b8e584152e4da/bridge/bridge/chain_sol.go#L72-L74

```
if fee < 50000000 {
    panic(fmt.Sprintf("sol fee paid to small to cover transaction cost: %d", fee))
}
```

In contrast, when locking or burning tokens in the EVM Bridge, users do not incur any additional fees. As a result, when users bridge tokens from EVM to Solana, the protocol must cover the transaction fees required to initiate the transactions for minting or unlocking tokens on Solana.

https://github.com/runemine/bridge/tree/76ca43cb0a519c81b0045efede9b8e584152e4da/bridge/contract-evm/Bridge.sol#L104-L124

```
function burn(
    string calldata name,
    uint16 targetNetwork,
    string calldata to,
    uint256 amount
) external {
    address token = tokens[name];
    Token(token).burn(msg.sender, amount);
    emit Burn(token, msg.sender, name, targetNetwork, to, amount);
}

// transfer in a token which is native to this chain, emitting an event the bridge
↪   will notice
function lock(
    address token,
    uint16 targetNetwork,
    string calldata to,
    uint256 amount
) external {
    Token(token).transferFrom(msg.sender, address(this), amount);
    emit Lock(token, msg.sender, targetNetwork, to, amount);
}
```

This vulnerability can be exploited by attackers, potentially leading to significant financial losses for the protocol through repeated minimal burns or locks.

Internal pre-conditions

none

External pre-conditions

none

Attack Path

Let's consider the following scenario:

1. Alice, the attacker, selects a target chain with the highest transaction fees for minting or unlocking tokens.
2. Alice repeatedly burns or locks dust amounts in the EVM Bridge.

As a result, Alice's burns and locks will trigger the protocol to initiate transactions to mint or unlock the corresponding tokens on the target chain, leading to significant transaction fees incurred by the protocol.

Impact

Loss of funds for the protocol.

PoC

none

Mitigation

A mechanism for covering gas fees should be implemented in the EVM Bridge.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/21

Issue H-5: Nonce management vulnerability leading to DoS in bridge transactions

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/22

The protocol has acknowledged this issue.

Found by

KupiaSec

Summary

The `evmSend` function in the bridge implementation is vulnerable to a Denial of Service (DoS) attack due to improper nonce management. When a transaction is front-run and reverts, the `lastNonce` variable is not updated, causing subsequent transactions to reuse an already used nonce. This leads to repeated failures, effectively halting the bridge's functionality for the affected network.

Root Cause

1. Improper Nonce Handling in `evmSend`:

   - The `lastNonce` for a target network is not updated when a transaction fails.

   https://github.com/sherlock-audit/2024-11-runemine/blob/main/bridge/bridge/chain_evm.go#L106

   - Subsequent transactions reuse the stale `lastNonce`, leading to reverts due to the contract's `nonces` check.

   https://github.com/sherlock-audit/2024-11-runemine/blob/main/bridge/bridge/chain_evm.go#L88-L91

2. Contract-Level Nonce Validation:

   - The bridge contract marks nonces as used after a successful or front-run transaction and update highest nonce.

   https://github.com/sherlock-audit/2024-11-runemine/blob/main/bridge/contract-evm/Multisig.sol#L60-L62

   - Reuse of a nonce results in reversion due to the contract's `require(!nonces[nonce])` check.

   https://github.com/sherlock-audit/2024-11-runemine/blob/main/bridge/contract-evm/Multisig.sol#L58

3. Faulty Logic in Nonce Calculation:
    - The condition `BiGte(BiFirst(lastNonce[transfer.Target]),nonce)` assumes `lastNonce` is always up-to-date, which is not guaranteed after a failure.

   https://github.com/sherlock-audit/2024-11-runemine/blob/main/bridge/bridge/chain_evm.go#L89

Internal Pre-Conditions

- `evmSend` relies on `lastNonce` to calculate the next nonce.
- Transactions that fail due to front-running do not update `lastNonce`.

External Pre-Conditions

- An attacker front-runs the transaction by preemptively using the calculated nonce.
- The transaction reverts, leaving `lastNonce` stale.

Attack Path

1. The attacker observes the bridge transaction being broadcasted with a specific nonce.
2. The attacker front-runs the transaction by submitting their own with the same nonce.
3. The bridge transaction fails and does not update `lastNonce`.
4. Subsequent transactions from the bridge reuse the same nonce, repeatedly reverting.
5. The bridge becomes stuck, unable to process further transactions for the affected network.

Impact

This vulnerability allows a malicious actor to halt all bridge transactions for a specific EVM network, effectively rendering the bridge inoperable for that network. The issue constitutes a Denial of Service (DoS) attack, impacting users and disrupting cross-chain functionality.

Proof of Concept (PoC)

1. Set up a bridge instance and initiate a transfer to an EVM network.

2. Front-run the bridge transaction by submitting a transaction with the same nonce as the bridge.
3. Observe that the bridge transaction reverts due to the nonce being already used.
4. Initiate another transfer. Observe that the same nonce is reused, causing repeated reverts.
5. The bridge becomes non-functional for the target network.

Mitigation

1. Ensure `lastNonce` is Always Updated:

   - Update `lastNonce[transfer.Target]` immediately after fetching the `highestNonce`, even in the event of a failure.

   Example Fix:

   ```
   defer func() {
       if err := recover(); err != nil {
           nonce := evmCall(rpc, "", msAddress,
    ↪  "highestNonce--uint256")[0].(*big.Int)
           lastNonce[transfer.Target] = nonce
           log.Println("Synchronized nonce after panic:", nonce)
       }
   }()
   ```

2. Use `highestNonce` Directly from the Contract:

   - Replace reliance on `lastNonce` with `highestNonce` from the contract to ensure nonce accuracy.

   Fix Example:

   ```
   nonce := evmCall(rpc, "", msAddress, "highestNonce--uint256")[0].(*big.Int)
   nonce = BiAdd(nonce, IntToBi(1)) // Always use the next available nonce
   ```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/18

Issue H-6: Bypassing Multisig Execution Threshold with Duplicate Signatures

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/27

Found by

Atharv, KupiaSec, cergyk

Summary

The protocol implements a multisig mechanism requiring at least 2 out of 3 admin signatures (2/3) to execute a transaction. However, the implementation can be exploited by submitting duplicate signatures in the array, allowing a transaction to execute with only one admin's signature.

Root Cause

The code does not ensure uniqueness in the signatures array during validation. As a result, the loop increments the counter n for each signature, even if the same signature is repeated, allowing the require(n >= execN) check to pass incorrectly.

```
for (uint i = 0; i < signatures.length; i++) {
    if (exec[recover(hash, signatures[i])]) {
        n++;
    }
}
require(n >= execN, "not enough signatures");
```

call function

And the call function is external hence anyone can call the function. Hence break the invarient that states it require atleast 2 admins to sign the payload to execute the transaction.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

1. Transactions can be executed with fewer signatures than the required threshold, compromising the security guarantees of the multisig mechanism.
2. A single malicious or compromised admin can exploit this to execute unauthorized transactions.
3. The flaw entirely defeats the purpose of using multisig for enhanced transaction security.

PoC

No response

Mitigation

To fix this vulnerability, ensure that each element in the signatures array is unique

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/16

Issue H-7: Cross-Chain Signature Replay Vulnerability in Multisig Contract

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/29

Found by

0xeix, Atharv, KupiaSec

Summary

The protocol is deployed across multiple EVM-compatible chains (Bitlayer, Botanix, Core, and Ethereum) with separate instances of Multisig.sol and Bridge.sol. The Multisig.sol contract uses the call() function to execute transactions authenticated by admin signatures. However, the payload for these signatures is not chain-specific, making the same signature valid across different chains.

This vulnerability allows an attacker to replay a valid signature from one chain to execute unauthorized transactions on another chain

Root Cause

The payload hash used for signature creation and verification does not include chain-specific data, making signatures chain-agnostic:

```
bytes32 payload = keccak256(abi.encode(nonce, target, value, data));
bytes32 hash = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32",
 ↪  payload));
```

This design flaw allows a signature created on one chain to be reused for transaction execution on another chain.

Payload Signature Verification

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Valid signatures are used by Bridge to call `call()` function on chain A.
2. Attacker sees the transaction data, signatures.
3. Use same signature and data on chain B.

Impact

1. Unauthorized transactions can be executed across chains using the same signatures.
2. Loss of funds if the target address on another chain contains different logic.
3. Unauthorized minting or unlocking of assets on other chains.

PoC

No response

Mitigation

To prevent signature replay across chains, follow the EIP-712 standard for signature creation and verification, which includes chain-specific data(chainId / targetNetwork).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/17/

Issue M-1: btc fee miscalculation may lead to transfer failures out of bridge

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/10

Found by

cergyk

Summary

When calling `btcSignAndSend` to send tokens to users when target network is bitcoin, the transaction fee is evaluated based on `inputs` and `outputs`, unfortunately the output struct size is not correclty accounted for. This may lead to insufficient fee provided for a transaction to be included, and either Dos or unexpected delays.

Root Cause

util_btc.go#L87:

```
txSize := int64(10 + (len(tx.TxIn) * 297) + (len(outputs) + 1*32))
```

We can see that `len(outputs)` is not multiplied by output structure length in bytes which we can assume to be `32`.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Some outbound transfers (from the bridge) may fail unexpectedly

PoC

No response

Mitigation

Consider modifying the txSize calculation:

util_btc.go#L87:

```
-   txSize := int64(10 + (len(tx.TxIn) * 297) + (len(outputs) + 1*32))
+   txSize := int64(10 + (len(tx.TxIn) * 297) + ((len(outputs) + 1)*32))
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/runemine/bridge/pull/19

Issue M-2: If utxo split for rune transfer happens near window end, amount transferred is incomplete

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/15

The protocol has acknowledged this issue.

Found by

cergyk

Summary

Rune transfers can be split into multiple edicts, and if that happens they will be indexed as separate transfers for the same transaction. Runemine handles this by aggregating them back into one amount by `tx` and `rune`, but if the query window is full (2500), some of those transfers may be truncated and left out.

Root Cause

The amounts are aggregated. but only for one query window:

chain_btc.go#L16-L17:

```
transfers := NJ(HttpVal("GET", Env("INDEXER", "")+"/transfers?limit=2500&to="+ms,
↪  "", nil)).GetA("transfers")
for _, transfer := range transfers {
```

chain_btc.go#L57-L62:

```
total := StringToBi("0")
for _, t2 := range transfers {
    if t2.Get("tx") == transfer.Get("tx") && t2.Get("rune") == transfer.Get("rune")
↪   {
        total = total.Add(total, StringToBi(t2.Get("amount")))
    }
}
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

User loses part of their funds

PoC

No response

Mitigation

Please consider handling the edge case when transfers for the same transaction are split between two indexer query windows

Issue M-3: Too high timeout value for HTTP request may cause missed transactions on Solana network because of low amount of fetched transactions

Source: https://github.com/sherlock-audit/2024-11-runemine-judging/issues/49

The protocol has acknowledged this issue.

Found by

TessKimy, cergyk, dod4ufn

Summary

Too high timeout value for HTTP request may cause missed transactions on Solana network because of low amount of fetched transaction

Root Cause

In util.go::HTTP() function, the timeout value is set to 90 seconds for HTTP calls. But this value is too high for error handling situations. In any packet loss situation this timeout value will stop the scanning for 90 seconds.

In 90 seconds, it's possible to miss transactions on Solana Network because Solana has a really high BPS rate and 90 seconds is really high timeout value for Solana Network.

```
func init() {
&>  http.DefaultClient.Timeout = 90 * time.Second
}
func Http(v interface{}, method, url, bodyString string, headers map[string]string)
 ↪  error { // @audit missing timeout for HTTP - high
    req, err := http.NewRequest(method, url, bytes.NewBufferString(bodyString))
    if err != nil {
        return err
    }
    req.Header.Set("Accept", "application/json")
    if headers != nil {
        for k, v := range headers {
            req.Header.Set(k, v)
        }
    }
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return fmt.Errorf("Http: call %s: %w", url, err)
    }
    defer resp.Body.Close()
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return fmt.Errorf("Http: read %s: %w", url, err)
    }
    if resp.StatusCode < 200 || resp.StatusCode >= 300 {
        panic(fmt.Errorf(`fetch: code %d: %s: %s`, resp.StatusCode, url,
 ↪  string(body)))
    }
    return json.Unmarshal(body, &v)
}

func (b *Bridge) solScan() []*Transfer {
    // fetch recent transactions that interracted with our program
    transfers := []*Transfer{}
&>  signatures := solanaRpc("getSignaturesForAddress", Env("SOLANA_PROGRAM", ""),
 ↪  J{"limit": 100}).([]interface{})
    for _, rs := range signatures {
        id := NJ(rs).Get("signature")
        if b.DB.Where("id = ?", id).First(&Transfer{}).Error == nil {
            continue // skip out early to save on requests
        }
        transfers = append(transfers, b.solScanTx(id)...)
    }
    return transfers
}
```

External pre-conditions

1. Timeout situation is happened (such as packet loss)

Attack Path

This situation can occur in high volatility moments.

1. There are 100 transactions in corresponding Solana Program address
2. Those transactions are fetched by the off-chain protocol
3. Timeout situation is happened
4. Another 200 transactions is submitted by the users
5. Solana executed those transactions before timeout ends
6. Now 100 transactions are lost because off-chain system will detect the last 100 transactions

Impact

High - Those transactions can't be detected by the protocol and they can't even solve the problem using the admin panel because the transactions aren't detected and stored in DB

The users lost their funds.

Mitigation

Apply reasonable amount of timeout for HTTP and use a bigger limit for Solana signature fetch.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.