Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Review-Report BOB Onramp Smart Contracts 04.2024

Cure53, Dr.-Ing. M. Heiderich, Dr. A. Pirker, Dr. N. Kobeissi

Index

Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

“BOB (Build on Bitcoin) is the first Bitcoin L2 with full EVM compatibility & native Bitcoin support empowering everyone to build and innovate on Bitcoin. BOB (Build on Bitcoin) enables DeFi and innovation across all fields of Bitcoin use cases & experimentation. Whatever you're building on Bitcoin, BOB is your swiss-army-knife for all things build on Bitcoin.”

From https://www.gobob.xyz/

This report, assigned the unique identifier *BOB-02-WP2*, presents the results of a Cure53 cryptography review and source code audit against the Onramp smart contracts, and associated codebases.

Stakeholders from Distributed Crafts Ltd. contacted Cure53 in April 2024 to discuss the aims and expected outcomes of the project. Once the scope and budget had been finalized, the review was scheduled for CW16 of the same month. Three senior pentesters from Cure53’s talent pool were selected to complete the assignment, based on their proficiency and experience handling components of this nature.

To aid the white-box technical analysis, Cure53 was provided with sources and other necessary materials. One distinct Work Package (WP) was created for efficiency reasons, defined by the following headings:

- **WP2**: Cryptography reviews & code audits against BOB Onramp codebase & SCs

Please note that the first work package (WP1) tracked within this project covered a different area of code and is documented in a separate report.

A number of essential preparations were completed in April 2024, namely in CW15, to encourage a seamless working environment. Throughout the assessment, communication channels remained open via a dedicated Telegram channel shared by the development team and Cure53. All relevant personnel from both parties joined the channel and engaged in the collaborative process when required.

The scope definition was clearly mapped out and the test team was suitably equipped to conduct the initiatives. Cure53 provided regular status updates on the testing progress and associated findings, though live reporting was deemed unnecessary. Onto the findings: a total of six were identified following widespread coverage over the WP2 scope items. To break those down, four were categorized as security vulnerabilities and the other two pertained to lower risk, common weaknesses.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The provided source code proved resilient to a plethora of typical breach schemes, which confirms the development team's effectiveness in minimizing the attack surface and mitigating vulnerabilities within the assessed components. However, several of the findings represent valuable hardening recommendations.

Moving forward, the report presents a selection of key chapters for ease of reference. Firstly, the *Scope* clarifies the test setup and available materials. Next, the *Identified Vulnerabilities* and *Miscellaneous Issues* comprise all observed findings in ticket format. The tickets provide supporting information such as a technical rundown, Proof-of-Concept (PoC), affected code examples, and remediation advice.

To finalize the document, the *Conclusions* section summarizes Cure53's opinion of the scope's security performance by taking a closer look at the coverage and discoveries.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Cryptography reviews & code audits against BOB Solidity SCs & Rust codebase**
  - **WP2:** Cryptography reviews & code audits against BOB Onramp codebase & SCs
    - **Sources:**
      - https://github.com/bob-collective/bob-onramp
    - **Commit:**
      - 297eeb1faeb7387bab142b4d2e0bcc6ef191747a
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, all tickets are given a unique identifier (e.g., *BOB-02-003*) to facilitate any future follow-up correspondence.

BOB-02-003 WP2: Integer overflow in block stream observer *(Low)*

Cure53 detected the presence of an integer overflow vulnerability in the *wait_for_block* function within Onramp's high-level Rust interface. This function waits until a specified block height achieves a minimum number of block confirmations before the block is accepted.

The vulnerability arises due to the insecure casting of the *num_confirmations* variable, which represents an unsigned 32-bit integer (*u32*) to a signed 32-bit integer (*i32*).

In scenarios whereby *num_confirmations* is excessively large (e.g., the maximum value of a *u32* - 2^{32-1} - or close to it), the cast into an *i32* can result in negative values due to an integer overflow. Consequently, the comparison logic might incorrectly evaluate to *true* even when the actual number of confirmations is insufficient, thus compromising the integrity of the block acceptance process.

Affected file:
bob-onramp/app/src/bitcoin_client.rs

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
async fn wait_for_block(&self, height: u32, num_confirmations: u32) ->
Result<Block, Error> {
        loop {
            match self.rpc.get_block_hash(height.into()) {
                Ok(hash) => {
                    let info = self.rpc.get_block_info(&hash)?;
                    if info.confirmations >= num_confirmations as i32 {
                        return Ok(self.rpc.get_block(&hash)?);
                    } else {
                        tokio::time::sleep(RETRY_DURATION).await;
                        continue;
                    }
                }
                Err(BitcoinError::JsonRpc(JsonRpcError::Rpc(err)))
                    if BitcoinRpcError::from(err.clone())
                        == BitcoinRpcError::RpcInvalidParameter =>
                {
                    // block does not exist yet
                    tokio::time::sleep(RETRY_DURATION).await;
                    continue;
                }
                Err(err) => {
                    return Err(err.into());
                }
            }
        }
    }
```

To mitigate this issue, Cure53 recommends altering the code to avoid unsafe casting throughout the confirmations calculation process. Alternatively, bounds can be implemented in order to ensure that the number of confirmations requested is never unreasonably low or high.

BOB-02-004 WP2: DoS in block stream observer *(Low)*

The *wait_for_block* function in the Onramp's high-level Rust interface exhibits a design paradigm that could lead to a Denial-of-Service (DoS) situation. This function is designed to fetch a block at a given height with a minimum number of confirmations from a blockchain node via JSON-RPC.

The concern revolves around the use of an indefinite loop that only exists under certain conditions, which could be manipulated or delayed indefinitely by an attacker, leading to resource exhaustion. The loop continuously polls the blockchain node for a block at a specified height with sufficient confirmations. If the block is not yet available or does not meet the confirmation threshold, the function sleeps for a predetermined duration (*RETRY_DURATION*) and then retries.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This loop can potentially run indefinitely if:

- The specified block height does not exist or is far into the future.
- The block at the specified height is withheld from achieving the necessary confirmations due to network issues or malicious activity.

Each iteration of the loop involves network calls that consume resources and possibly CPU cycles on the server hosting the blockchain node.

Affected file:
bob-onramp/app/src/bitcoin_client.rs

Affected code:

```
async fn wait_for_block(&self, height: u32, num_confirmations: u32) ->
Result<Block, Error> {
        loop {
            match self.rpc.get_block_hash(height.into()) {
                Ok(hash) => {
                    let info = self.rpc.get_block_info(&hash)?;
                    if info.confirmations >= num_confirmations as i32 {
                        return Ok(self.rpc.get_block(&hash)?);
                    } else {
                        tokio::time::sleep(RETRY_DURATION).await;
                        continue;
                    }
                }
                Err(BitcoinError::JsonRpc(JsonRpcError::Rpc(err)))
                    if BitcoinRpcError::from(err.clone())
                        == BitcoinRpcError::RpcInvalidParameter =>
                {
                    // block does not exist yet
                    tokio::time::sleep(RETRY_DURATION).await;
                    continue;
                }
                Err(err) => {
                    return Err(err.into());
                }
            }
        }
    }
```

To mitigate the risk of a DoS situation and enhance the robustness of the *wait_for_block* function, the developer team should consider the following modifications:

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- **Timeout implementation:** Introduce a maximum timeout for the loop to ensure that the function does not run indefinitely. This could be implemented as an elapsed time or maximum number of retries.
- **Rate limiting:** Incorporate a rate limit for the number of times the function can query the blockchain within a certain time period.
- **Exponential backoff:** Rather than leverage a fixed sleep interval, one could integrate an exponential backoff mechanism for the sleep duration between retries. This strategy would reduce the load on both the network and server during periods of high demand or failure.

BOB-02-009 WP2: Gas waste and potential griefing via gratuity & fee *(Low)*

Fix note: *The issue was mitigated by the customer during the assessment and fix-verified by Cure53.*

Liquidity Providers (LPs) correspond to the owners of Onramp contracts. In the Onramp system, a relayer creates a new Onramp contract for an LP through the *OnrampFactory* smart contract. The relayer provides the LP address to the *OnrampFactory* contract that creates the new instance of the *Onramp* contract, which in turn sets the LP as owner. The LP of the Onramp contract can adjust certain parameters, such as the gratuity or fee divisor, by first invoking the *startUpdate* function. After a six-hour delay, the LP may adjust the parameters arbitrarily. Finally, the LP ends the parameter update by invoking the *endUpdate* function. With this in mind, Cure53 confirmed that the parameters for the gratuity and fee divisor of an Onramp contract are not checked prior to being configured, meaning that arbitrary values are permitted.

This enables a malicious LP to set the values for gratuity and fee divisor to problematic numbers. For instance, when the *OnrampFactory* contract executes a swap, the Onramp contract ultimately transfers a gratuity to the swap's *_recipient*. Setting the gratuity explicitly to 0 will not transfer any funds and essentially wastes gas. Furthermore, the LP could also set the fee divisor to 1, which implies that the entire amount intended to be transferred corresponds to the fee. This results in a 0 amount transfer on the utilized ERC20 token. Depending on the ERC20 token implementation, this results in a revert. However, a waste of gas for the operator executing the swap will be induced in all cases.

Notably, this issue could be exploited via swap execution front-running due to the flawed update guard described in ticket BOB-02-007, which amplifies this attack vector. Specifically, a relayer could execute swaps even after the update security window, which in turn facilitates executing both *executeSwap* and functions that use the modifier *canUpdate*, such as *setGratuity*. However, the team was also not able to identify a guard off-chain to prevent arbitrary values, since the gratuity and fee divisor off-chain to automatically block the swap execution after an update process was concluded by an LP.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The snippet below demonstrates the flaw for the Onramp smart contract's *gratuity* field, whereby one can verify that the field is never checked for the 0 value.

Affected file #1:
bob-onramp/contracts/src/Onramp.sol

Affected code #1:
```
constructor(
    [...]
    uint64 _gratuity
) Ownable() {
    [...]
    gratuity = _gratuity;
}
[...]
function setGratuity(uint64 _gratuity) external onlyOwner canUpdate {
    emit UpdateGratuity(_gratuity);
    gratuity = _gratuity;
}
[...]
function executeSwap(
    bytes32 _txHash,
    uint256 _outputValueSat,
    address payable _recipient
) external onlyFactory {
    [...]
    _recipient.transfer(gratuity);
}
```

The snippet below highlights the issue for the Onramp smart contract's *feeDivisor* field, whereby one can also confirm that this field is never checked for the 1 value. Furthermore, one can deduce from the *calculateFee* function that a *feeDivisor* of 1 corresponds to the full amount of the swap fee.

Affected file #2:
bob-onramp/contracts/src/Onramp.sol

Affected code #2:
```
constructor(
    [...]
    uint64 _feeDivisor,
    [...]
) Ownable() {
    [...]
    feeDivisor = _feeDivisor;
    gratuity = _gratuity;
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
}
[...]
function setFeeDivisor(uint64 _feeDivisor) external onlyOwner canUpdate {
    emit UpdateFeeDivisor(_feeDivisor);
    feeDivisor = _feeDivisor;
}
[...]
function executeSwap(
    bytes32 _txHash,
    uint256 _outputValueSat,
    address payable _recipient
) external onlyFactory {
    [...]
    uint256 feeSat = calculateFee(_outputValueSat);
    uint256 amount = calculateAmount(_outputValueSat - feeSat);

    emit ExecuteSwap(_recipient, _outputValueSat, feeSat, amount,
gratuity);

    // transfer token
    require(token.transfer(_recipient, amount), "Could not transfer
ERC20");
    [...]
}
[...]
function calculateFee(uint256 _outputValueSat)
    [...]
{
    uint256 feeSat = feeDivisor > 0 ? _outputValueSat / feeDivisor : 0;
    return feeSat;
}
```

To mitigate this issue, Cure53 advises checking both the *gratuity* and *feeDivisor* fields for plausible values on all assignments, specifically 0 and 1, respectively.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

BOB-02-010 WP2: Permanent DoS of Onramp contract by LP *(Medium)*

Cure53 observed that the BOB Onramp system involves several parties. Some parties, such as the relayer, may be considered trusted whereas others, like Liquidity Providers (LPs), are assumed to act maliciously. In this threat model, Cure53 identified multiple attacks whereby LPs could evoke a permanent Denial-of-Service (DoS) situation against an Onramp contract instance. A permanent DoS via a maliciously acting LP corresponds to a waste of gas for all contracts owned by the LP in question, as the operator of the relayer created them and must now accommodate for the blocked Onramp contract instances by creating new Onramp contract instances with other LPs.

For example, a malicious LP may initiate an update process by invoking the *startUpdate* function on an Onramp contract instance owned by the provider. After six hours, i.e. the update delay, the relayer cannot execute any swaps via this Onramp contract instance, due to the (fixed) update guard mentioned in ticket BOB-02-007. Until the LP invokes the *endUpdate* function, the Onramp contract instance will no longer be accessible for the relayer. This constitutes a permanent DoS for the Onramp contract instance, as only the LP itself may unblock the Onramp contract instance via the *endUpdate* function.

Additional scenarios that could exacerbate this vulnerability entail the following:

- A malicious LP could coordinate initiating the update process across multiple Onramp contract instances simultaneously, which would maximize disruption by affecting multiple contracts and increasing the impact and visibility of the attack.

- By manipulating economic incentives, a malicious LP could render continued use of the affected Onramp contract instances unprofitable for relayers. This could involve artificially increasing the costs associated with transactions or creating market conditions that enhance the viability of alternatives. The flaw described in ticket BOB-02-009 could also magnify this attack strain.

- Repeated attacks (or even simply the threat of them) could damage the reputation of the Onramp system. Over time, this could lead to a decrease in trust among users and other LPs, potentially reducing the overall user base and volume of transactions.

To mitigate this issue, Cure53 advises reviewing the roles and responsibilities of Onramp contracts. The internal team could enforce that trusted relayers remain owners of Onramp contracts and can modify the associated LP. As such, trusted relayers will be able to intervene should malicious LPs become active.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

BOB-02-005 WP2: Block timestamp dependence of update modifier *(Info)*

The Onramp contract contains a modifier that prevents calling certain functions, while a Liquidity Provider (LP) alters contract fields/parameters. The LP, who corresponds to the owner of Onramp contracts, must explicitly initiate an update by calling the *startUpdate* function, which sets the *updateStart* field to the *block.timestamp* value. Commencing an update gives the relayer time to execute all pending orders before the LP is able to modify contract parameters. Here, testing verified that the fields involved in either allowing or blocking functions during an update process depend on a block's timestamp, i.e. *block.timestamp*.

Continued use of a block's timestamp could potentially incur risk from a security perspective, since miners can influence it in a malicious manner to evoke potential DoS situations, among other plausible implications[1].

One can deduce from the code excerpt below that the Onramp contract employs the value of *block.timestamp* to mark the start of an update.

Affected file:
bob-onramp/contracts/src/Onramp.sol

Affected code:
```
function startUpdate() external onlyOwner {
    updateStart = uint64(block.timestamp);
    emit StartUpdate(updateStart);
}
```

To mitigate this issue, Cure53 discourages relying on the value of *block.timestamp* to mark the start of an update. Alternatively, the implementation should use off-chain and trusted oracles to provide a timestamp externally.

[1] https://gloryprаise.hashnode.dev/vulnerability-8-timestamp-dependence-vulnerability

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

BOB-02-007 WP2: Insufficient update guard for swap execution *(Info)*

Fix note: *The issue was mitigated by the customer during the assessment and fix-verified by Cure53.*

The Onramp contract contains a guard statement to block swaps while an update of the contract's parameters is in progress. This guard statement serves to protect the swap initiator against a malicious LP (as the owner of the Onramp contracts) that intentionally raises fees, modifies other parameters, or even withdraws funds by front-running the swap invoked via the *OnrampFactory*. Here, the team confirmed that the implemented guard statement is always *true* and therefore does not offer any protection against a malicious LP.

Further discussions with the customer confirmed that this limitation is already known. Therefore, this ticket's severity score was downgraded from *High* to *Info* and transferred to the miscellaneous rather than vulnerabilities section.

The code snippet below highlights the guard statement. Note that the *updateStart* field is set to *block.timestamp* (via the *startUpdate* function), which renders the guard condition always *true*.

Affected file:
bob-onramp/contracts/src/Onramp.sol

Affected code:

```
function executeSwap(
    bytes32 _txHash,
    uint256 _outputValueSat,
    address payable _recipient
) external onlyFactory {
        // slither-disable-next-line timestamp
        require(
            updateStart <= block.timestamp + UPDATE_DELAY,
            "Not allowed to execute"
        );

    [...]
}
```

To mitigate this issue, Cure53 advises revising the guard statement. The condition should correspond to *block.timestamp* <= *updateStart* + *UPDATE_DELAY*, since this allows the swap execution before the *canUpdate* modifier (utilized for the functions modifying the contract parameters) becomes effective.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

This Q2 2024 audit comprised two work packages. The one covered in this report, WP2, concentrated on reviewing the Onramp smart contracts and associated Rust codebase. The private repository targeted for WP2 was made accessible to Cure53 in advance. Since this assignment constituted a source code audit exclusively, additional assets such as an infrastructure or test environment were not provided.

The external and in-house teams remained in contact via a specifically established Telegram channel, which hosted open questions and allowed the testers to relay progress updates. The cross-team communication was generally excellent and assistance was provided whenever requested.

The auditors achieved an ample degree of coverage within the allotted time frame. In context, the source code of both work packages was compositionally moderate. The smart contracts are written in Solidity while the off-chain aspect of WP2 is written in Rust. Positively, the respective code bases were well organized upon inspection.

The smart contracts were reviewed for common vulnerabilities that affect Solidity specifically:

- The first area of concern was reentrancy issues. Cure53 found that the smart contracts perform external calls almost exclusively after performing all state-changing operations to the contract itself, which tends to rule out reentrancy flaws by default. Despite strenuous efforts in this area, the test team was unable to discover any connected problems.

- Oftentimes, Solidity contracts suffer from arithmetic errors due to loss of precision, resulting from an incorrect order of arithmetic operations. Furthermore, former versions of Solidity neglect to check for overflow and underflow situations. Nevertheless, the assessors verified that the smart contracts exhibit negligible attack surface with regards to these circumstances.

- The team also stringently investigated the visibilities and modifiers of the contract functions. However, the conclusion was made that the smart contracts do not expose any mechanisms that would otherwise widen the attack surface, contributing to the robust overall impression.

- Another focus aspect was the likelihood of DoS situations and griefing attacks, which could be attempted by threat actors in order to disrupt or modify the behavior of smart contracts. The team explored this vulnerability angle in depth, discovering several points of contention. A malicious LP could either render Onramp smart contracts permanently redundant (see ticket BOB-02-010) or modify parameters

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

such as gratuity or fees, therefore wasting operator gas or even preventing them from further Onramp use (as highlighted in ticket BOB-02-009).

- Cure53 also determined that the Onramp contract implements a guard condition prior to executing swaps. The update mechanism leverages the *block.timestamp* value; this is generally discouraged for security sensitive operations since miners may influence the value, as discussed in ticket BOB-02-005. The update guard intends to protect relayers from executing swaps while an LP modifies contract parameters, owing to the fact that LPs could modify parameters in a malicious and possibly covert fashion (see BOB-02-009). Moreover, testing identified that the update guard was faulty and permitted swap execution while updates were in progress (see BOB-02-007). Similarly to deficiencies described in the previous paragraph, the developer team has already taken swift action to remediate this.

- Elsewhere, the audit team searched for missing authorization checks and attacks leading to impersonations. Here, it was positively concluded that the contracts successfully nullify these compromise strategies. Cure53 also sought to pinpoint any logical flaws such as the assignment of absolute approval values, for instance, though no associated behaviors were noted.

Moving on, the off-chain Rust codebase was subjected to rigorous examinations by the test team:

- The Rust API and Bitcoin client frontend was assessed by focusing on two key targets: the correctness of cryptography-relevant operations and the security/soundness of the exposed server API.

- As part of the Rust cryptography analysis, heightened emphasis was placed on guaranteeing the correctness of the Merkle tree implementation logic and Bitcoin chain querying operations. Cure53's endeavors revealed two minor detriments in the block stream observer logic, as indicated in tickets BOB-02-003 and BOB-02-004.

- Cure53's vetting procedures against the Rust API code was ultimately unfruitful. The API was considered minimal and self-contained, while no obvious avenues for third-party exploitation were detected. Nevertheless, as with any API, security against abuse or DoS depends on optimally gated deployment and access controls.

- The application employs a PostgreSQL database for persistence, thus the testers estimated the application's susceptibility to SQL injection vulnerabilities. Positively, the implementation was deemed risk averse in this respect.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- Cure53 also investigated whether the application exposes potential sinks for Remote Code Execution (RCE). One sole instance of this was observed concerning the Bitcoin daemon, though the test team was unable to exploit this sink via a malicious actor.

In summary, Cure53 can confirm that the provided source code exhibits satisfactory security proficiency under the current configuration. Many of the identified issues correspond to hardening recommendations that will provide defense-in-depth and further protect assets against malicious actors outside of the current threat model. The development team has successfully minimized the exposed attack surface and negated most vulnerability classes that could plausibly affect the characteristics in-scope for this audit. Lastly, the in-house team's diligence toward addressing some of the pressing concerns soon after detection is commendable and corroborates the argument that their framework is progressing in an upward trajectory from a security viewpoint.

Cure53 would like to thank Gregory Hill, Sander Bosma, and Dominik Harz from the Distributed Crafts Ltd. team for their excellent project coordination, support, and assistance, both before and during this assignment.