SOFTWARE AUDIT REPORT

for

BHOP CONSULTANTING PTE. LTD.

Prepared By: Shuxiao Wang

Hangzhou, China

August 15, 2020

Document Properties

Client	BHOP Consultanting Pte. Ltd.
Title	Software Audit Report
Target	HBTC Chain
Version	1.0
Author	Xuxian Jiang
Auditors	Ruiyi Zhang, Edward Lo, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	August 15, 2020	Xuxian Jiang	Final Release
1.0-rc	August 8, 2020	Xuxian Jiang	Release Candidate
0.4	July 9, 2020	Xuxian Jiang	More Findings #4
0.3	July 2, 2020	Xuxian Jiang	More Findings #3
0.2	June 24, 2020	Xuxian Jiang	More Findings #2
0.1	June 16, 2020	Xuxian Jiang	Initial Report #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **HBTC Chain** design document and related source code, we in this report outline our systematic method to evaluate potential security issues in the HBTC Chain implementation, expose possible semantic inconsistencies between the source code and the design specification, and provide additional suggestions and recommendations for improvement. Our results show that the given branch of HBTC Chain can be further improved due to the presence of several issues related to either security or performance. This document describes our audit results in detail.

1.1 About HBTC Chain

HBTC Chain presents the next-generation blockchain-based technology for decentralized asset custody and clearing. Capitalizing on recent development of various technologies (e.g., threshold cryptography and blockchain), and further integrating with wide community-based decentralized consensus, HBTC Chain greatly advances the solutions to address a variety of security and trust problems faced by many traditional centralized digital asset platforms.

The basic information of HBTC Chain is shown in Table 1.1, and its Git repository and the commit hash value (of the audited branch) are in Table 1.2.

Table 1.1: Basic Information of HBTC Chain

Item	Description
Issuer	BHOP Consultanting Pte. Ltd.
Website	https://chain.hbtc.com/
Type	HBTC Chain
Coding Language	Go
Audit Method	White-box
Latest Audit Report	August 15, 2020

Table 1.2: The Commit Hash List Of Repositories or Branches For Audit

Git Repository	Commit Hash	Coverage
https://github.com/hbtc-chain/bhchain.git	344dfc1	Yes
https://github.com/hbtc-chain/settle.git	3852ef8	Yes
https://github.com/hbtc-chain/dsign.git	1576b56	Yes
https://github.com/hbtc-chain/chainnode.git	f1f55e0	Yes
https://github.com/hbtc-chain/tendermint.git	e412b2a	No
https://github.com/hbtc-chain/crypto.git	1b9364b	No
https://github.com/hbtc-chain/sssa-golang.git	3ff434d	No

1.2 About PeckShield

PeckShield Inc. [1] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products including security audits. We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

In the first phase of auditing HBTC Chain, we use fuzzing to find out the corner cases that may not be covered by in-house testing. However, our major efforts are in white-box auditing, in which PeckShield security auditors manually review HBTC Chain design and source code, analyze them for any potential issues, and follow up with issues found in the fuzzing phase. If necessary, we design and implement individual test cases to further reproduce and verify the issues. In the following subsections, we will introduce the risk model as well as the audit procedure adopted in this report.

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [2]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, and *Low* shown in Table 1.3.

1.3.2 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by systematically finding and providing possible inputs to the target program, and then monitoring the program execution for crashes (or any unexpected results). In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing. As one of the most effective methods for exposing the presence of possible vulnerabilities, fuzzing technology has been the first choice for many security researchers in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to conduct fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the HBTC Chain, we use AFL [3] as the primary tool for fuzz testing.

AFL (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Since its inception, AFL has gained growing popularity in the industry and has proved its effectiveness in discovering quite a few significant software bugs in a wide range of major software projects. The basic process of AFL fuzzing is as follows:

- Generate compile-time instrumentation to record information such as code execution path;
- Construct some input files to join the input queue, and change input files according to different strategies;

- Files that trigger a crash or timeout when executing an input file are logged for subsequent analysis;

- Loop through the above process.

Throughout the AFL testing, we will reproduce each crash based on the crash file generated by AFL. For each reported crash case, we will further analyze the root cause and check whether it is indeed a vulnerability. Once a crash case is confirmed as a vulnerability of the HBTC Chain, we will further analyze it as part of the white-box audit.

1.3.3 White-box Audit

After fuzzing, we continue the white-box audit by manually analyzing source code. Here we test target software's internal structure, design, coding, and we focus on verifying the flow of input and output through the application as well as examining possible design and implementation trade-offs for strengthened security. PeckShield auditors first fully review and understand the source code, then create specific test cases, execute them and analyze the results. Issues such as internal security loopholes, unexpected output, broken or poorly structured paths, etc., will be inspected under close scrutiny.

Blockchain is a secure method of creating a distributed database of transactions, and three major technologies of blockchain are cryptography, decentralization, and consensus model. Blockchain does come with unique security challenges, and based on our understanding of blockchain general design, we in this audit divide the blockchain software into the following major areas and inspect each area accordingly:

- Data and state storage, which is related to the database and files where blockchain data are saved.

- P2P networking, consensus, and transaction model in the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.

- VM, account model, and incentive model. This is essentially the execution and business layer of the blockchain, and many blockchain business specific logics are implemented here.

- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.

- SDK Security. These are include additional SDK modules and example code for developer community distribution and adoption.

Table 1.4: The Full List of Audited Items (Part I)

Category	Check Item
Data and State Storage	Blockchain Database Security
	Database State Integrity Check
Node Operation	Default Configuration Security
	Default Configuration Optimization
	Node Upgrade And Rollback Mechanism
Node Communication	External RPC Implementation Logic
	External RPC Function Security
	Node P2P Protocol Implementation Logic
	Node P2P Protocol Security
	Serialization/Deserialization
	Invalid/Malicious Node Management Mechanism
	Communication Encryption/Decryption
	Eclipse Attack Protection
	Fingerprint Attack Protection
Consensus	Consensus Algorithm Scalability
	Consensus Algorithm Implementation Logic
	Consensus Algorithm Security
Transaction Model	Transaction Privacy Security
	Transaction Fee Mechanism Security
	Transaction Congestion Attack Protection
VM	VM Implementation Logic
	VM Implementation Security
	VM Sandbox Escape
	VM Stack/Heap Overflow
	Contract Privilege Control
	Predefined Function Security
Account Model	Status Storage Algorithm Adjustability
	Status Storage Algorithm Security
	Double Spending Protection
Incentive Model	Mining Algorithm Security
	Mining Algorithm ASIC Resistance
	Tokenization Reward Mechanism

Table 1.5: The Full List of Audited Items (Part II)

Category	Check Item
System Contracts And Services	Memory Leak Detection
	Use-After-Free
	Null Pointer Dereference
	Undefined Behaviors
	Deprecated API Usage
	Signature Algorithm Security
	Multisignature Algorithm Security
	Nervos DAO Mechanism
SDK Security	Using RPC Functions Security
	PrivateKey Algorithm Security
	Communication Security
	Function integrity checking code
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

- Others. This includes any software modules that do not belong to above-mentioned areas, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Based on the above classification, we show in Table 1.4 and Table 1.5 the detailed list of the audited items in this report.

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [4], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.6 to classify our findings.

Table 1.6: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of blockchain software. Last but not least, this security audit should not be used as an investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing HBTC Chain. As mentioned earlier, we in the first phase of our audit studied HBTC Chain source code (including related libraries) and ran our in-house static code analyzer through the codebase, and we focused on `bhchain`, `bhsettle`, and `chainnode`. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tools. After that, we manually review business logic, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

Table 2.1: The Severity of Our Findings

Severity	# of Findings	
Critical	0	
High	4	■■■■
Medium	8	■■■■■■■■
Low	5	■■■■■
Informational	5	■■■■■
Total	22	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple modules. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined several issues of varying severities that need to be brought up and paid more attention to. These issues are categorized in the above Table 2.1. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, the HBTC Chain is well-designed and engineered, though the implementation can be improved by resolving the identified issues (as shown in Table 2.2), including 4 high-severity vulnerabilities, 8 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.2: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incomplete Genesis State For Future Upgrades	Init. & Cleanup	Fixed
PVE-002	Medium	Inappropriate Initialization Order of Modules	Init. & Cleanup	Fixed
PVE-003	Medium	Free Key Generation in handleMsgKeyGen()	Business Logic	Fixed
PVE-004	Medium	Improper Fee Return in handleMsgKeyGenFinish()	Business Logic	Fixed
PVE-005	High	Asset Lockdown on OpcuAssetTransfer()	Time and State	Fixed
PVE-006	High	Unintended Deposit Removal on OpcuAssetTransfer()	Time and State	Fixed
PVE-007	Low	Improved Precision Price Calculation in OpcuAssetTransferWaitSign()	Numeric Errors	Fixed
PVE-008	Low	Inaccurate Sufficiency Calculation in OpcuAssetTransferWaitSign()	Business Logic	Fixed
PVE-009	Informational	Improved Necessity Checks in SysTransfer()	Business Logics	Fixed
PVE-010	Informational	Generation of Meaningful Events for MsgSend/MsgMultiSend	Coding Practices	Fixed
PVE-011	Low	Tolerant Error-Handling in ConfirmedDeposit()	Business Logic	Fixed
PVE-012	Low	Proper AssetCoins Reduction For Dust Deposits	Business Logic	Fixed
PVE-013	High	Possible Flooding Attacks in handleMsgOrderRetry()	Security Features	Fixed
PVE-014	Informational	Invalid Order Removal in handleMsgDeposit()	Security Features	Confirmed
PVE-015	Medium	Missing Error Handling in SignedTx Verification	Coding Practices	Fixed
PVE-016	Informational	Blackhole Receipt Addresses of MsgSend/MsgMultiSend	Coding Practices	Fixed
PVE-017	Medium	Unrecognized Contract-Sourced ETH Deposits in Chainnode	Business Logic	Confirmed
PVE-018	Informational	Slashing Non-Cooperating Members in Key Management	Business Logic	Fixed
PVE-019	Medium	Proper Safe Prime Generation	Coding Practices	Fixed
PVE-020	Medium	Unconstrained Private Key Range in sssa.Create()	Arguments and Parameters	Fixed
PVE-021	Low	Zeroizing Secret Temporary Values	Coding Practices	Fixed
PVE-022	Medium	Missing Validity Check in MtAwc	Security Features	Confirmed

3 Detailed Results

3.1 Incomplete Genesis State For Future Upgrades

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `cu, token, hrc20, order, ...`
- Category: Initialization & Cleanup [5]
- CWE subcategory: CWE-459 [6]

Description

HBTC Chain is developed on top of Cosmos-SDK[7], a popular modular framework for building application-specific blockchains. Note that Cosmos-SDK enables rapid development of SDK-based blockchains out of composable modules. HBTC Chain leverages some existing modules (with its own customization) and further develops unique ones for the purpose of digital asset custody and clearing. Specifically, `cu` provides a custody unit for asset management, `token` enlists available tokens for trading or custody, `mapping` supports cross-chain asset mapping, `keygen` provides dynamic key generation services for cross-chain assets, and `hrc20` enables ERC20-like token issuance and transfer primitives on HBTC Chain etc.

The modular Cosmos-SDK framework allows various modules to generally handle a subset of the state and, as such, these modules need to define the related subset of the genesis file as well as methods to initialize, verify, and export it. We stress that these states are essential to the blockchain's genesis state import and export and are therefore required for seamless upgrades. In the current HBTC Chain codebase, several modules do not have thorough genesis-related states properly exported or imported. Consequently, they could lead to broken upgrades.

Using the `custodianunit` (a.k.a., `cu`) module as an example, we show below the implementation of current `InitGenesis` and `ExportGenesis` routines. As the names indicate, they are executed whenever an import or export of the state is made. The `ExportGenesis` routine exported both `params` and `cus`, but the `InitGenesis` routine only imported `params`, not `cus`. In other words, all created `cus` in a previous run may be lost for a resumed run of HBTC Chain after upgrade.

```go
// InitGenesis - Init store state from genesis data
//
// CONTRACT: old coins from the FeeCollectionKeeper need to be transferred through
// a genesis port script to the new fee collector CU
func InitGenesis(ctx sdk.Context, ak CUKeeper, data GenesisState) {
    ak.SetParams(ctx, data.Params)
}

// ExportGenesis returns a GenesisState for a given context and keeper
func ExportGenesis(ctx sdk.Context, ck CUKeeper) GenesisState {
    params := ck.GetParams(ctx)
    cus := ck.GetAllCUs(ctx)
    return NewGenesisState(params, cus)
}
```

Listing 3.1: bhchain/x/custodianunit/genesis.go

Meanwhile, it is important to note that the validation of `ValidateGenesis` is also relatively incomplete, without the validation of saved `cus` in the genesis state.

```go
// ValidateGenesis performs basic validation of auth genesis data returning an
// error for any failed validation criteria.
func ValidateGenesis(data GenesisState) error {
    if data.Params.TxSigLimit == 0 {
        return fmt.Errorf("invalid tx signature limit: %d", data.Params.TxSigLimit)
    }
    if data.Params.SigVerifyCostED25519 == 0 {
        return fmt.Errorf("invalid ED25519 signature verification cost: %d", data.Params.SigVerifyCostED25519)
    }
    if data.Params.SigVerifyCostSecp256k1 == 0 {
        return fmt.Errorf("invalid SECK256k1 signature verification cost: %d", data.Params.SigVerifyCostSecp256k1)
    }
    if data.Params.MaxMemoCharacters == 0 {
        return fmt.Errorf("invalid max memo characters: %d", data.Params.MaxMemoCharacters)
    }
    if data.Params.TxSizeCostPerByte == 0 {
        return fmt.Errorf("invalid tx size cost per byte: %d", data.Params.TxSizeCostPerByte)
    }
    return nil
}
```

Listing 3.2: bhchain/x/custodianunit/types/genesis.go

Similar genesis-related issues are also present in other modules, including `token`, `hrc20`, `order`, `transfer`, and `keygen`. And their individual routines in (`InitGenesis`, `ExportGenesis`, and `ValidateGenesis`) also need to be revised accordingly.

Recommendation Appropriately import and export necessary genesis state in affected modules.

3.2 Inappropriate Initialization Order of Modules

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `keygen`, `supply`, `distr`, ...
- Category: Initialization & Cleanup [5]
- CWE subcategory: CWE-459 [6]

Description

The various composable modules in HBTC Chain have their own internal dependencies that need to be honored during initialization and tear-down. For example, the module `genutils` must occur after `staking` so that pools are properly initialized with tokens from genesis accounts. Also, `capability` module, if present, must occur first so that it can initialize any capabilities so that other modules that want to create or claim capabilities afterwards can do so safely. Similarly, `gov` and `slashing` depend on `bank` for accessing and/or modifying balances.

```go
	// During begin block slashing happens after distr.BeginBlocker so that
	// there is nothing left over in the validator fee pool, so as to keep the
	// CanWithdrawInvariant invariant.
	app.mm.SetOrderBeginBlockers(mint.ModuleName, distr.ModuleName, slashing.ModuleName)
	app.mm.SetOrderEndBlockers(crisis.ModuleName, gov.ModuleName, staking.ModuleName)

	// NOTE: The genutils moodule must occur after staking so that pools are
	// properly initialized with tokens from genesis accounts.
	app.mm.SetOrderInitGenesis(
		genaccounts.ModuleName, otypes.ModuleName, receipt.ModuleName, token.ModuleName, keygen.ModuleName, distr.ModuleName, staking.ModuleName,
		custodianunit.ModuleName, transfer.ModuleName, slashing.ModuleName, gov.ModuleName,
		mint.ModuleName, supply.ModuleName, crisis.ModuleName, genutil.ModuleName, hrc20.ModuleName, mapping.ModuleName,
	)
```

Listing 3.3: bhchain/bhexapp/app.go

An examination of the current codebase, as shown in the above snippet, reveals inconsistent initialization of included modules. In particular, the `app.mm.SetOrderInitGenesis()` indicates the `InitGenesis` order that needs to satisfy inherent dependency. In Figure 3.1, we outline the actual dependency among current modules in HBTC Chain. The actual dependency is derived by examining the related inter-module keeper references in each module. In other words, if a module, say `keygen`, references `order`, we can infer `keygen` depends on `order`, hence `order -> keygen` in the dependency figure.

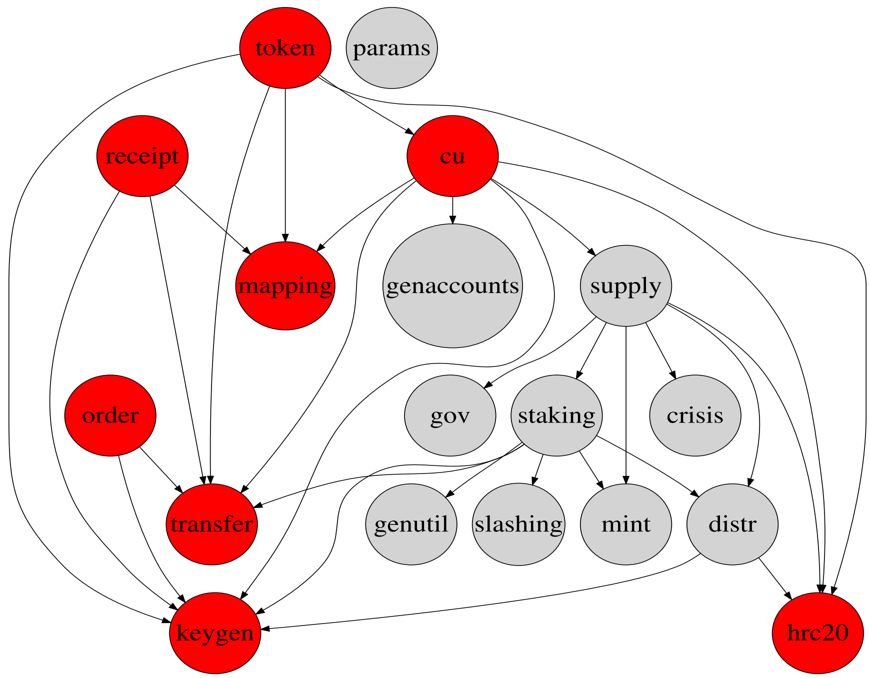

Figure 3.1: The Module Dependency in HBTC Chain

By cross-checking `app.mm.SetOrderInitGenesis` with the above dependency figure, we obtain the following possible violations of module dependency: `keygen`, `distr`, `staking`, `cu`, `transfer`, `gov`, `mint`, and `supply`. An inappropriate order may cause broken initialization or introduce wrong runtime state and thus need to be certainly avoided.

Recommendation The relevant fixup is rather straightforward. Basically, we need to apply the module initialization in the order by following their inherent dependency.

```

    // NOTE: The genutils moodule must occur after staking so that pools are
    // properly initialized with tokens from genesis accounts.
    app.mm.SetOrderInitGenesis(

    /* Old Order
        genaccounts.ModuleName, otypes.ModuleName, receipt.ModuleName, token.ModuleName,
             keygen.ModuleName, distr.ModuleName, staking.ModuleName,
        custodianunit.ModuleName, transfer.ModuleName, slashing.ModuleName, gov.
            ModuleName,
        mint.ModuleName, supply.ModuleName, crisis.ModuleName, genutil.ModuleName, hrc20
            .ModuleName,
        mapping.ModuleName,
    */

        // New Order
        otypes.ModuleName, receipt.ModuleName, token.ModuleName,
```

```
        custodianunit.ModuleName,   genaccounts.ModuleName,    supply.ModuleName,
        gov.ModuleName, staking.ModuleName, crisis.ModuleName, slashing.ModuleName,
        genutil.ModuleName, mint.ModuleName, distr.ModuleName,
        transfer.ModuleName, keygen.ModuleName, hrc20.ModuleName,
        mapping.ModuleName,
    )
```

Listing 3.4: bhchain/bhexapp/app.go (revised)

3.3 Free Key Generation in handleMsgKeyGen()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `keygen`
- Category: Business Logic [8]
- CWE subcategory: CWE-666 [9]

Description

Among all modules in HBTC Chain, the `keygen` module is an important one. This particular module provides dynamic key generation services for cross-chain assets. Specifically, when a user requests for the creation of a custody address in a supported external chain (via a `MsgKeyGen` message), `keygen` processes the request in the `handleMsgKeyGen` handler by essentially delegating the request to the `settle` daemon.

Importantly, this module processes `MsgKeyGen` messages via the `handleMsgKeyGen` handler, which differentiates three types of scenarios: `subToken`, `WaitAssignKeyGenOrder`, and `KeyGenOrder`. The first `subToken` scenario indicates the request for an address of an ERC20-like asset. The second `WaitAssignKeyGenOrder` scenario examines the presence of a pre-generated address and, if any, directly allocates one to answer the request. The third `KeyGenOrder` scenario leaves the heavy-lifting task of custody address key generation to `settle`.

The second scenario shares an issue that allows for free key generation. Specifically, the associated opening fee `feeCoins` has not been deducted from the requesting user – `fromCU`, though the same amount has been credited to `CommunityPool` (via the `keeper.dk.AddToFeePool()` in line 158 in the following code snippet).

```
151         ...
152         flows := make([]sdk.Flow, 0, 3+len(keygenOrder.KeyNodes))
153         orderflow := keeper.rk.NewOrderFlow(symbol, fromAddr, orderID, sdk.
                OrderTypeKeyGen, sdk.OrderStatusFinish)
154         keyGenFinishFlow := sdk.KeyGenFinishFlow{OrderID: orderID, IsPreKeyGen: true, To
                : toAddr}
155         flows = append(flows, orderflow, keyGenFinishFlow)
```

```
        if feeCoins.IsAllGT(sdk.NewCoins(sdk.NewCoin(sdk.NativeToken, sdk.ZeroInt()))) {
            keeper.dk.AddToFeePool(ctx, sdk.NewDecCoins(feeCoins))
        }
    ...
```

Listing 3.5: bhchain/x/keygen/handler.go

Recommendation The fixup is straightforward as we need to properly charge the key-generation opening fee from the requesting `fromCU` onto the storage in second scenario: `WaitAssignKeyGenOrder`.

3.4 Improper Fee Return in handleMsgKeyGenFinish()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `keygen`
- Category: Business Logic [8]
- CWE subcategory: CWE-399 [10]

Description

As described in Section 3.1, the `keygen` module provides dynamic key generation services for cross-chain assets. Specifically, it has a few handlers to process various types of messages. In the previous sections, we have been focusing on the `handleMsgKeyGen()` handler that processes `MsgKeyGen` messages. In this section, we examine another handler, i.e., `handleMsgKeyGenFinish()`, which processes `MsgKeyGenFinish` messages. As the name indicates, these messages notify the HBTC Chain that previous requests for key generation have been finished.

There is an issue inside `handleMsgKeyGenFinish()` that improperly returns back the opening fee back to the requesting users. Moreover, it further disseminates the same amount of opening fee to the `distr` module, resulting in an unintended inflation on `hbc` - the HBTC Chain native token.

Specifically, we show below the related code snippet inside the `handleMsgKeyGenFinish()` handler. After performing necessary sanity checks on the `MsgKeyGenFinish` message, the system eventually charges the opening fee by moving the amount of opening fee (previously on hold on `fromCU.SubCoinsHold`) to `distr`. However, the line 305 indicates that the opening fee is also returned back to `fromCU`.

```
298     ...
299     //sub openfee
300     fromCU := keeper.ck.GetCU(ctx, fromCUAddr)
301     openFee := keyGenOrder.OpenFee
302     hasFee := openFee.IsAllGT(sdk.NewCoins(sdk.NewCoin(sdk.NativeToken, sdk.ZeroInt())))
303     if hasFee {
```

```
            fromCU.SubCoinsHold(openFee)
            fromCU.AddCoins(openFee)
            keeper.ck.SetCU(ctx, fromCU)
            keeper.dk.AddToFeePool(ctx, sdk.NewDecCoins(openFee))
        }
        ...
```

Listing 3.6: bhchain/x/keygen/handler.go

Recommendation There is no need to return back the key-generation opening fee back to the requesting `fromCU`.

```
        ...
        //sub openfee
        fromCU := keeper.ck.GetCU(ctx, fromCUAddr)
        openFee := keyGenOrder.OpenFee
        hasFee := openFee.IsAllGT(sdk.NewCoins(sdk.NewCoin(sdk.NativeToken, sdk.ZeroInt())))
        if hasFee {
            fromCU.SubCoinsHold(openFee)
            // fromCU.AddCoins(openFee)
            keeper.ck.SetCU(ctx, fromCU)
            keeper.dk.AddToFeePool(ctx, sdk.NewDecCoins(openFee))
        }
        ...
```

Listing 3.7: bhchain/x/keygen/handler.go (revised)

3.5 Asset Lockdown in OpcuAssetTransfer()

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High

- Target: `transfer`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [12]

Description

Among all modules in HBTC Chain, the `transfer` module is one of the most crucial modules and its main functionality is to actually transfer assets inside and outside HBTC Chain. Its complexity is also partially reflected in the number of messages it recognizes and handles. In total, there are 21 different types of messages, including `MsgSend`, `MsgDeposit`, `MsgWithdraw`, `MsgSysTransfer`, `MsgOpcuAssetTransfer`, and their variants.

In this section, we mainly focus on the four specific message types related to `opcu` asset transfers: i.e., `MsgOpcuAssetTransfer`, `MsgOpcuAssetTransferWaitSign`, `MsgOpcuAssetTransferSignFinish`, and

MsgOpcuAssetTransferFinish. Among these four message types, the first one – MsgOpcuAssetTransfer – aims to kick off the process for transferring assets under opcu custody; the second one – MsgOpcuAssetTransferWaitSign – is responsible for starting the key-signing process on behalf of opcu and the keys are shared among active validators of HBTC Chain; the third one – MsgOpcuAssetTransferSignFinish – signals the accomplishment of key-signing process so that the signed transaction can be broadcasted and mined on chain; and the fourth one – MsgOpcuAssetTransferFinish – reaches the consensus in successfully completing the transfer transaction and therefore properly settles down necessary asset updates after the transfer.

If we delve into the MsgOpcuAssetTransfer-handling logic, this specific handler OpcuAssetTransfer takes a few arguments that specify the related opcu account opCUAddr, the new destination address toAddr, the asset symbol symbol, as well as related TransferItems included in this particular transfer. To facilitate the organization and management of the entire transfer process, HBTC Chain has its internal order system. And each particular transfer has its unique orderID. Different transfers will have different orderIDs.

However, this particular handler suffers from an issue that may be abused to lock down the funds being transferred. Specifically, it does NOT validate the freshness of the given destination address toAddr to ensure it is generated for current migration epoch. As a result, an outdated toAddr can be provided to bypass the following check (line 60).

```
53      ...
54      valid, canonicalToAddr := keeper.cn.ValidAddress(chain, symbol, toAddr)
55      if !valid {
56          return sdk.ErrInvalidAddr(fmt.Sprintf("%v is not a valid address", toAddr)).
                Result()
57      }
58
59      toAsset := opCU.GetAssetByAddr(symbol, canonicalToAddr)
60      if toAsset == sdk.NilAsset {
61          return sdk.ErrInvalidAddr(fmt.Sprintf("%v does not belong to cu %v",
                canonicalToAddr, opCU.GetAddress().String())).Result()
62      }
63      ...
```

Listing 3.8: bhchain/x/transfer/keepers/opcuasset_transfer.go

After the check, the handler further discerns two different token types: Utxo and Account. The first type relates to assets in BTC and the second type is about assets in Ethereum. For the BTC assets, each TransferItem will be accordingly marked as DepositItemStatusInProcess. As a result, the asset may not be released until the transfer process completes. For Ethereum assets, the opcu account will be marked as opCU.SetEnableSendTx(false, chain, fromAddr), which prevents any asset from being transferred under its custody unless the flag is turned back to true.

```
        ...
        for _, item := range items {
```

```
            depositItem := keeper.ck.GetDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
            if depositItem == sdk.DepositNil || !depositItem.Amount.Equal(item.Amount) ||
                depositItem.Status == sdk.DepositItemStatusInProcess {
                return sdk.ErrInvalidTx(fmt.Sprintf("Invalid DepositItem(%v)", item.Hash)).Result()
            }
            sum = sum.Add(item.Amount)
        }
        ...
        for _, item := range items {
            keeper.ck.SetDepositStatus(ctx, symbol, opCUAddr, item.Hash, item.Index, sdk.DepositItemStatusInProcess)
        }
        ...
```

Listing 3.9: bhchain/x/transfer/keepers/opcuasset_transfer.go

If the above crafted `opcu` asset transfer process completes, the assets will be transferred to the old `toAddr`, which no one further holds the secret shares. If the transfer process does not complete, the internal states have been modified in a way that prevents the same `TransferItems` from being re-used or `opCU` from being transferable for assets under its custody. In either way, the funds are locked from future use.

Recommendation Add additional sanity checks to ensure the freshness of `toAddr`, i.e. it is the latest one being generated for current `epoch`.

3.6 Unintended Deposit Removal in OpcuAssetTransfer()

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `transfer`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [12]

Description

As mentioned earlier, the `transfer` module is one of the most crucial modules and its main functionality is to allow for asset transfers inside and outside HBTC Chain. In the last section, we have examined a vulnerability related to `MsgOpcuAssetTransfer` handling. In this section, we examine another issue within the same handler that could lead to unintended removal of legitimate deposits.

Specifically, this `OpcuAssetTransfer` handler takes a few arguments that specify the related `Opcu` account `opCUAddr`, the new destination address `toAddr`, the asset symbol `symbol`, as well as related

`TransferItems` included in this particular transfer. To facilitate the organization and management of the entire transfer process, HBTC Chain has its internal `order` system. And each particular transfer has its unique `orderID`. Different transfers will have different `orderIDs`.

The last issue deals with the non-freshness of one argument – `toAddr`. This issue is related to duplicability in another argument – `TransferItems`. We show below the message type `MsgOpcuAssetTransfer`'s validity check routine: `ValidateBasic()`. Apparently, there is a check on `TransferItems`. But it only ensures that the `TransferItems` array is not empty. In other words, it does not check whether any item in the array is duplicated or not. As a result, we can construct a message type with duplicates in `TransferItems`.

```go
// quick validity check
func (msg MsgOpcuAssetTransfer) ValidateBasic() sdk.Error {
    // note that unmarshaling from bech32 ensures either empty or valid
    _, err := sdk.CUAddressFromBase58(msg.FromCU)
    if err != nil {
        return ErrBadAddress(DefaultCodespace)
    }

    _, err = sdk.CUAddressFromBase58(msg.OpCU)
    if err != nil {
        return ErrBadAddress(DefaultCodespace)
    }

    if msg.ToAddr == "" {
        return ErrBadAddress(DefaultCodespace)
    }

    if msg.OrderID == "" {
        return ErrNilOrderID(DefaultCodespace)
    }

    if len(msg.TransferItems) == 0 {
        return sdk.ErrInvalidTx("transfer items are empty")
    }

    return nil
}
```

Listing 3.10: bhchain/x/transfer/types/msgs.go

Further, assume there is a particular `TransferItem` A whose amount is less than `OpcuAstTransferThreshold`. (If there is none, we can always create one.) In the meantime, there is another `TransferItem` B with amount above `OpcuAstTransferThreshold`. For simplicity, `A.amount=0.1*OpcuAstTransferThreshold`, and `B.amount=1.1*OpcuAstTransferThreshold`. Accordingly, we can construct an array with two As and one B such that the `sum` of them meets the conditional check in line 98, i.e., `sum.LTE(keeper.utxoOpcuAstTransferThreshold(len(items), tokenInfo))`. Note that the right-end value grows linearly with the number of items in `TransferItems`, which implies it is always feasible to construct such

TransferItems.

```
        ...
        sum := sdk.ZeroInt()
        for _, item := range items {
            depositItem := keeper.ck.GetDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
            if depositItem == sdk.DepositNil || !depositItem.Amount.Equal(item.Amount) ||
                depositItem.Status == sdk.DepositItemStatusInProcess {
                return sdk.ErrInvalidTx(fmt.Sprintf("Invalid DepositItem(%v)", item.Hash)).Result()
            }
            sum = sum.Add(item.Amount)
        }

        if sum.LTE(keeper.utxoOpcuAstTransferThreshold(len(items), tokenInfo)) {
            for _, item := range items {
                keeper.ck.DelDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
            }
            burnedCoins := sdk.NewCoins(sdk.NewCoin(symbol, sum))
            opCU.AddGasUsed(burnedCoins)
            keeper.ck.SetCU(ctx, opCU)
            if keeper.checkUtxoOpcuAstTransferFinish(ctx, fromAddr, symbol, opCU) {
                opCU.SetMigrationStatus(sdk.MigrationFinish)
                keeper.ck.SetCU(ctx, opCU)
                keeper.checkOpcusMigrationStatus(ctx, curEpoch)
            }
            return sdk.Result{}
        }
        ...
```

Listing 3.11: bhchain/x/transfer/keepers/opcuasset_transfer.go

Once the condition has been met, the deposit item related to B will be considered too small and is then "safely" deleted, causing possible loss on assets under opcu custody. In addition, it also messes up the internal GasUsed states.

Recommendation Add additional sanity checks to ensure the uniqueness of TransferItems.

3.7 Improved Precision Price Calculation in OpcuAssetTransferWaitSign()

- ID: PVE-007
- Severity: low
- Likelihood: Medium
- Impact: Low
- Target: `transfer`
- Category: Numeric Errors [13]
- CWE subcategory: CWE-190 [14]

Description

As mentioned in Section 3.5, `Opcu` asset transfers require the handling of four specific message types: i.e., `MsgOpcuAssetTransfer`, `MsgOpcuAssetTransferWaitSign`, `MsgOpcuAssetTransferSignFinish`, and `MsgOpcuAssetTransferFinish`. Among these four message types, the first one – `MsgOpcuAssetTransfer` – aims to kick off the process for transferring assets under `Opcu` custody; the second one – `MsgOpcuAssetTransferWaitSign` – is responsible for starting the key-signing process on behalf of `Opcu` and the keys are shared among active validators of HBTC Chain; the third one – `MsgOpcuAssetTransferSignFinish` – signals the accomplishment of key-signing process so that the signed transaction can be broadcasted and mined on chain; and the fourth one – `MsgOpcuAssetTransferFinish` – reaches the consensus in successfully completing the transfer transaction and therefore properly settles down necessary asset updates after the transfer.

When handling the second message type, i.e., `MsgOpcuAssetTransferWaitSign`, there is a need to calculate the transaction price. However, it is currently calculated in way that may lead to precision loss (line 246): `sdk.NewDecFromInt(tx.CostFee).Quo(size).MulInt64(sdk.KiloBytes)`.

```
243         ...
244         //Estimate SignedTx Size and calculate price
245         size := sdk.EstimateSignedUtxoTxSize(len(tx.Vins), len(tx.Vouts)).ToDec()
246         price := sdk.NewDecFromInt(tx.CostFee).Quo(size).MulInt64(sdk.KiloBytes)
247 
248         if price.GT(priceUpLimit) {
249             return sdk.ErrInvalidTx(fmt.Sprintf("gas price is too high, actual:%v,
                uplimit:%v", price, priceUpLimit)).Result()
250         }
251         if price.LT(priceLowLimit) {
252             return sdk.ErrInvalidTx(fmt.Sprintf("gas price is too low, actual:%v,
                lowlimit:%v", price, priceLowLimit)).Result()
253         }
254         ...
```

Listing 3.12: bhchain/x/transfer/keepers/opcuasset_transfer.go

For improved precision, it is suggested to calculate the multiplication before the division, i.e., `sdk.NewDecFromInt(tx.CostFee).MulInt64(sdk.KiloBytes).Quo(size)`.

There is another similar issue in the handling of message type `MsgCollectWaitSign` in function `CollectWaitSign` (line 104) that can also benefit from the above calculation with improved precision.

Recommendation Revise the above calculation to better handle possible precision loss: i.e., `sdk.NewDecFromInt(tx.CostFee).MulInt64(sdk.KiloBytes).Quo(size)`.

3.8 Inaccurate Sufficiency Calculation in OpcuAssetTransferWaitSign()

- ID: PVE-008
- Severity: low
- Likelihood: Medium
- Impact: Low
- Target: `transfer`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [15]

Description

In this section, we examine the same handling logic `MsgOpcuAssetTransferWaitSign` as in Section 3.7, and expose another logic issue.

Specifically, before kicking off the key-signing process, we need to ensure the sender has the sufficient funds available to pay the transaction amount, plus the associated gas fee. In the following code snippet, we highlight the related variables and their calculation. It becomes evident that when `order.Symbol == chain` (line 289) holds, the gas fee has been accounted for in `tx.Amount`. The follow-up addition of `coins = coins.Add(feeCoins)` basically counts the gas fee twice: both `tx.GasPrice.Mul(tokenInfo.GasLimit)` and `tx.CostFee`.

```
288         ...
289         if order.Symbol == chain {
290             tx.Amount = tx.Amount.Add(tx.GasPrice.Mul(tokenInfo.GasLimit))
291         }
292         ...
293         feeCoins := sdk.NewCoins(sdk.NewCoin(chain, tx.CostFee))
294         coins := sdk.NewCoins(sdk.NewCoin(symbol, tx.Amount))
295         coins = coins.Add(feeCoins)
296         if !opCU.GetAssetCoins().IsAllGTE(coins) {
297             return sdk.ErrInsufficientCoins(fmt.Sprintf("opCU has insufficient coins,
                    expected: %v, actual have:%v", coins, opCU.GetAssetCoins())).Result()
298         }
299         ...
```

Listing 3.13: bhchain/x/transfer/keepers/opcuasset_transfer.go

Recommendation Correct the above calculation for the inclusion of associated gas fee.

3.9 Improved Necessity Checks in SysTransfer()

- ID: PVE-009
- Severity: Informational
- Likelihood: Medium
- Impact: N/A
- Target: `transfer`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [15]

Description

In this section, we examine a handling logic, i.e., `SysTransfer()`, for the message type `MsgSysTransfer`, which is used to fund necessary gas fee for internal collection of either external user deposits or internal `opcu` transfers.

Specifically, this `SysTransfer()` handler takes a few arguments that specify the fund source `fromCUAddr`, the destination `toCUAddr` and its external address `toAddr`, the asset symbol `symbol`, as well as the gas fee amount `amt` for this particular transfer. Similarly, to facilitate the organization and management of the entire transfer process, a unique `orderID` is chosen for this particular `SysTransfer`. Different transfers will have different `orderID`s.

```go
	...
	toCU := keeper.ck.GetCU(ctx, toCUAddr)
	if toCU == nil {
		return sdk.ErrInvalidAccount(toCUAddr.String()).Result()
	}
	valid, canonicalToAddr := keeper.cn.ValidAddress(chain, symbol, toAddr)
	if !valid {
		return sdk.ErrInvalidAddr(fmt.Sprintf("%v is not a valid address", toAddr)).Result()
	}
	toCUAsset := toCU.GetAssetByAddr(symbol, canonicalToAddr)
	if toCUAsset == sdk.NilAsset {
		return sdk.ErrInvalidTx(fmt.Sprintf("%v does not belong to cu %v", toAddr, toCU.GetAddress().String())).Result()
	}
	if toCU.GetCUType() == sdk.CUTypeOp && toCU.GetMigrationStatus() == sdk.MigrationFinish {
		if toCUAsset.Epoch != keeper.sk.GetCurrentEpoch(ctx).Index {
			return sdk.ErrInvalidTx("Cannot sys transfer to last epoch addr").Result()
		}
	}

	if keeper.hasProcessingSysTransfer(ctx, toCUAddr, chain, canonicalToAddr) {
		return sdk.ErrInvalidTx(fmt.Sprintf("To OPCU %v has processing sys transfer of %s", toCUAddr, chain)).Result()
	}
```

```
    ...
```

Listing 3.14: bhchain/x/transfer/keepers/systransfer.go

Before the `systransfer` process is initiated, it is necessary to ensure that `toAddr` is indeed in need of gas fee for future collection or transfers. If it does not hold any meaningful assets, there is no need to initiate the process at all. We do realize that an offline procedure can be used to ensure it, but it is always helpful to encode the logic within HBTC Chain to avoid unnecessary waste of gas fee, even the fee amount might not be significant.

Recommendation Apply additional checks to ensure the necessity of `toAddr` for gas fee.

3.10 Generation of Meaningful Events for MsgSend/MsgMultiSend

- ID: PVE-010
- Severity: Informational
- Likelihood: Medium
- Impact: N/A
- Target: `transfer`
- Category: Coding Practices [16]
- CWE subcategory: CWE-1116 [17]

Description

Event and logs are an important part of blockchain that can greatly facilitate encapsulation and expressiveness of blockchain activities and expose them to external monitoring DApps. For this end, it is always helpful to generate the events or logs in a way that are precise and expressive.

During the analysis of the `transfer` module (this is responsible for handling 21 message types), we notice that two message types handled by the module have not generated meaningful events. Specifically, these two message types, i.e., `MsgSend` and `MsgMultiSend`, if properly handled, generate events that simply bear with the generic event type – `sdk.EventTypeMessage`. This may not be considered expressive and it is thus strongly suggested to make their corresponding event types more specific and meaningful.

```
// Handle MsgSend.
func handleMsgSend(ctx sdk.Context, k keeper.Keeper, msg types.MsgSend) sdk.Result {
  if !k.GetSendEnabled(ctx) {
    return types.ErrSendDisabled(k.Codespace()).Result()
  }

  if k.BlacklistedAddr(msg.ToAddress) {
    return sdk.ErrUnauthorized(fmt.Sprintf("%s is not allowed to receive transactions",
        msg.ToAddress)).Result()
  }
```

```

    result, err := k.SendCoins(ctx, msg.FromAddress, msg.ToAddress, msg.Amount)
    if err != nil {
      return err.Result()
    }

    ctx.EventManager().EmitEvent(
      sdk.NewEvent(
        sdk.EventTypeMessage,
        sdk.NewAttribute(sdk.AttributeKeyModule, types.AttributeValueCategory),
      ),
    )

    result.Events = append(result.Events, ctx.EventManager().Events()...)
    return result
}
```

Listing 3.15: bhchain/x/transfer/handler.go

Recommendation Emit specific and meaningful events when handling `MsgSend` and `MsgMultiSend`
.

3.11 Tolerant Error-Handling in ConfirmedDeposit()

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `transfer`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [18]

Description

In this section, we revisit the `transfer` module and in particular examine the handling logic of `MsgConfirmedDeposit` messages. For each user deposit, it needs to be confirmed by a majority of current validators and each message can carry two arrays, one for valid deposits and another for invalid deposits. In other words, the message can batch-process a number of `orderIDs` simultaneously.

To elaborate, we show below the code snippet of one core helper routine, i.e., `processDepositOrderIDs` . Inside the routine, there is a `for` loop that basically iterates over each `orderID` and checks whether it has been confirmed. The confirmed `orderIDs` will be returned back the caller. However, we notice that if there is an error in handling a confirmed `orderID`, the rest `orderIDs` are simply ignored. The ignored `orderIDs` likely will go through another round of message submission and processing, delaying the entire deposit process. To avoid unnecessary delayed user experience, it is suggested to be more tolerant and continue processing rest `orderIDs` while simply ignoring error-causing `orderIDs`.

```go
func (keeper BaseKeeper) processDepositOrderIDs(ctx sdk.Context, fromAddr string, confirmThreshold int, orderIDs []string, valid bool) ([]sdk.Flow, []string, error) {
  var confirmedOrderIDs []string
  var flows []sdk.Flow
  for _, id := range orderIDs {
    order := keeper.ok.GetOrder(ctx, id)
    if order == nil || order.GetOrderType() != sdk.OrderTypeCollect {
      continue
    }
    collectOrder, ok := order.(*sdk.OrderCollect)
    if !ok {
      continue
    }
    nodes := collectOrder.InvalidNodes
    if valid {
      nodes = collectOrder.ValidNodes
    }
    if sdk.StringsIndex(nodes, fromAddr) >= 0 {
      continue
    }
    nodes = append(nodes, fromAddr)
    if valid {
      collectOrder.ValidNodes = nodes
    } else {
      collectOrder.InvalidNodes = nodes
    }
    if collectOrder.DepositStatus == sdk.DepositUnconfirm && len(nodes) >= confirmThreshold {
      balanceFlows, err := keeper.confirmDepositOrder(ctx, collectOrder, valid)
      if err != nil {
        return nil, nil, err
      }
      confirmedOrderIDs = append(confirmedOrderIDs, id)
      flows = append(flows, balanceFlows...)
    }

    keeper.ok.SetOrder(ctx, collectOrder)
  }
  return flows, confirmedOrderIDs, nil
}
```

Listing 3.16: bhchain/x/transfer/keeper/deposit.go

Recommendation The fixup is straightforward as we can choose not to `return`, but `continue` within the `for` loop.

3.12 Proper AssetCoins Reduction For Dust Deposits

- ID: PVE-012
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `transfer`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [18]

Description

When initiating asset transfers under `opcu` custody, there is a need to verify the transferred amount has reached certain threshold. For `Utxo`-based token types, the threshold is defined by `utxoOpcuAstTransferThreshold` while for `Account`-based token types, the threshold is defined by `tokenInfo.SysTransferAmount`. The purpose is to detect the necessity of not transferring assets only with dust amount. When such a dust amount is detected, HBTC Chain simply considers it burned and reduces the very same dust amount from owned `AssetCoins`.

Our analysis show that this is indeed the case for `Account`-based token types. However, for `Utxo`-based token types, the dust amount has been burned, but not reduced from the owned `AssetCoins` (as shown in line 103 in the code snippet below).

```
88	sum := sdk.ZeroInt()
89	for _, item := range items {
90		depositItem := keeper.ck.GetDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
91		if depositItem == sdk.DepositNil || !depositItem.Amount.Equal(item.Amount) ||
92			depositItem.Status == sdk.DepositItemStatusInProcess {
93			return sdk.ErrInvalidTx(fmt.Sprintf("Invalid DepositItem(%v)", item.Hash)).
				Result()
94		}
95		sum = sum.Add(item.Amount)
96	}
97
98	if sum.LTE(keeper.utxoOpcuAstTransferThreshold(len(items), tokenInfo)) {
99		for _, item := range items {
100			keeper.ck.DelDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
101		}
102		burnedCoins := sdk.NewCoins(sdk.NewCoin(symbol, sum))
103		opCU.AddGasUsed(burnedCoins)
104		keeper.ck.SetCU(ctx, opCU)
105		if keeper.checkUtxoOpcuAstTransferFinish(ctx, fromAddr, symbol, opCU) {
106			opCU.SetMigrationStatus(sdk.MigrationFinish)
107			keeper.ck.SetCU(ctx, opCU)
108			keeper.checkOpcusMigrationStatus(ctx, curEpoch)
109		}
110		return sdk.Result{}
111	}
```

Listing 3.17: bhchain/x/transfer/keeper/opcuasset_transfer.go

Recommendation Properly reduce the `AssetCoins` for dust deposits.

```
    if sum.LTE(keeper.utxoOpcuAstTransferThreshold(len(items), tokenInfo)) {
      for _, item := range items {
        keeper.ck.DelDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
      }
      burnedCoins := sdk.NewCoins(sdk.NewCoin(symbol, sum))
      opCU.SubAssetCoins(burnedCoins)
      opCU.AddGasUsed(burnedCoins)
      keeper.ck.SetCU(ctx, opCU)
      if keeper.checkUtxoOpcuAstTransferFinish(ctx, fromAddr, symbol, opCU) {
        opCU.SetMigrationStatus(sdk.MigrationFinish)
        keeper.ck.SetCU(ctx, opCU)
        keeper.checkOpcusMigrationStatus(ctx, curEpoch)
      }
      return sdk.Result{}
    }
```

Listing 3.18: bhchain/x/transfer/keeper/opcuasset_transfer.go (revised)

3.13 Possible Flooding Attacks in handleMsgOrderRetry()

- ID: PVE-013
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `transfer`
- Category: Security Features [19]
- CWE subcategory: CWE-284 [20]

Description

As discussed in earlier sections, the `transfer` module handles 21 different message types. In this section, we focus on the handling logic of one particular message type, i.e., `MsgOrderRetry`. Its sole purpose is to retry orders in case previous tries do no finish.

This message type has a field named `RetryTimes`. Our analysis shows that this field have not gone through rigorous checks. Because of that, it is possible to send a series of `MsgOrderRetry` messages with the same `OrderIDs` and `from` but with different `RetryTimes`. As a result, the handler proceeds the execution to a function called `addOrderRetryConfirmNode` (see the code snippet below), which keeps storing these confirmed messages into local `store`. Notice that the store key is calculated via `retryOrderKey(strings.Join(orderIDs, "&"), retrytimes)`. This indicates that key length can be arbitrarily controlled by user input, i.e., `OrderIDs`, and these occupied storage space will not be released forever. At the very least, it results in resource waste or inefficiency. When the accumulated waste occupies the full storage space, it eventually jeopardizes various normal chain-wide operations.

```go
func (keeper BaseKeeper) addOrderRetryConfirmNode(ctx sdk.Context, txID, validatorAddr
    string, retrytimes uint32, valsNum int) bool {
  retryOrderConfirmNodes := []string{}
  store := ctx.KVStore(keeper.storeKey)
  bz := store.Get(retryOrderKey(txID, retrytimes))
  if bz != nil {
    keeper.cdc.MustUnmarshalBinaryBare(bz, &retryOrderConfirmNodes)
  }

  bFind := false
  for _, v := range retryOrderConfirmNodes {
    if v == validatorAddr {
      bFind = true
      break
    }
  }

  if !bFind {
    retryOrderConfirmNodes = append(retryOrderConfirmNodes, validatorAddr)
    bz = keeper.cdc.MustMarshalBinaryBare(retryOrderConfirmNodes)
    store.Set(retryOrderKey(txID, retrytimes), bz)

    //have been confirmed
    if len(retryOrderConfirmNodes)-1 >= sdk.Majority23(valsNum) {
      return false
    }

    if len(retryOrderConfirmNodes) >= sdk.Majority23(valsNum) {
      return true
    }
  }

  return false
}
```

Listing 3.19: bhchain/x/transfer/keeper/utils.go

Recommendation Apply additional checks on `RetryTimes` in `MsgOrderRetry`.

3.14 Invalid Order Removal in handleMsgDeposit()

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `transfer`
- Category: Security Features [19]
- CWE subcategory: CWE-284 [20]

Description

In this section, we focus on the handling logic of another particular message type, i.e., `MsgDeposit`. Its main purpose is to record the occurrence of user deposits and, if needed, start the internal collection process.

The handling logic is implemented in the function `handleMsgDeposit()`. Within the function, we notice that several fields of `MsgDeposit`, including `OrderID`, `hash` and `index`, have not gone through rigorous checks. Because of that, an attacker can craft a flurry of `MsgDeposit` messages, each with a new different `OrderID` and a different `hash/index`. The handler continues the execution to the end by creating a flurry of `NewOrderCollect` orders and saving them into the `order` keeper.

```
	...
if keeper.ok.IsExist(ctx, orderID) {
	return sdk.ErrInvalidOrder(fmt.Sprintf("order %v already exists", orderID)).Result()
}

if keeper.ck.IsDepositExist(ctx, symbol.String(), toCUAddr, hash, index) {
	return sdk.ErrInvalidTx(fmt.Sprintf("deposit %v %v %v %v item already exist", symbol, toCU, hash, index)).Result()
}

//ProcessOrder should be optimized.
processOrderList := keeper.ok.GetProcessOrderListByType(ctx, sdk.OrderType_Collect)
for _, id := range processOrderList {
	order := keeper.ok.GetOrder(ctx, id)
	if order != nil {
		collectOrder := order.(*sdk.OrderCollect)
		if collectOrder.Txhash == hash && collectOrder.Index == index {
			return sdk.ErrInvalidTx(fmt.Sprintf("Tx: %v is already exist and not finish", hash)).Result()
		}
	}
}

collectOrder := keeper.ok.NewOrderCollect(ctx, toCUAddr, orderID, symbol.String(),
	toCUAddr, canonicalToAddr, amt, sdk.ZeroInt(), sdk.ZeroInt(), hash, index, memo)
keeper.ok.SetOrder(ctx, collectOrder)
```

Listing 3.20: bhchain/x/transfer/keeper/deposit.go

It is worth mentioning that the `transfer` module also processes `handleMsgConfirmedDeposit` messages. But, all invalid orders have not been deleted in the end, only their states have been updated to `Finish`. Therefore, at the very least, it results in resource waste or inefficiency. When the accumulated invalid `OrderIDs` occupy the full storage space, it eventually jeopardizes various normal chain-wide operations.

```
func (keeper BaseKeeper) confirmDepositOrder(ctx sdk.Context, order *sdk.OrderCollect, valid bool) ([]sdk.Flow, error) {
	var flows []sdk.Flow
```

```
order.DepositStatus = sdk.Deposit_Confirmed
if !valid {
  order.SetOrderStatus(sdk.OrderStatus_Finish)
  return flows, nil
}
```

Listing 3.21: bhchain/x/transfer/keeper/deposit.go

Recommendation Delete invalid orders regularly in `handleMsgConfirmedDeposit`.

3.15 Missing Error Handling in SignedTx Verification

- ID: PVE-015
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `transfer`
- Category: Coding Practices [16]
- CWE subcategory: CWE-1071 [21]

Description

Though the `transfer` module recognizes and processes 21 different types of messages, these messages typically revolves the business logic for user deposits, withdrawals, internal collection, `opcu` asset transfers, as well as gas fee transfer (for `Account`-based token types only). And the internal processing logic typically follows four different stages: `Begin`, `WaitSign`, `SignFinish`, and `Finish`. The first one indicates the intention to get started; the second one calls for the key-signing process and requires the responses from current validators; the third one finishes the key-signing process so that the signed transaction can be broadcasted and mined on chain; and the fourth one reaches the consensus in successfully completing the logic and therefore properly settles down necessary asset updates.

In this section, we examine a common routine, i.e., `verifyAccountBasedSignedTx`, often used in the third stage that verifies the key signature from current validators. This routine is important and required to block any malicious attempt to corrupt the key signing behavior. However, within the routine (see the code snippet below), it misses an error handling (line 112). We point out that the return value from calling the `chainnode.QueryAccountTransactionFromData(chain, symbol, rawData)` takes the untrusted input `rawdata` and should be thoroughly validated. In the case when the returned value – `rawTx` could be nil, the immediate access after the return will lead to a `null` pointer deference and crash the running process.

```
96 func (keeper BaseKeeper) verifyAccountBasedSignedTx(fromAddr, chain, symbol string,
     rawData, signedTx []byte) (sdk.Result, string) {
97   txHash := ""
98   verified, err := keeper.cn.VerifyAccountSignedTransaction(chain, symbol, fromAddr,
       signedTx)
```

```
    if err != nil || !verified {
        return sdk.ErrInvalidTx(fmt.Sprintf("VerifyAccountSignedTransaction fail:%v, err:%v", signedTx, err)).Result(), txHash
    }

    tx, err := keeper.cn.QueryAccountTransactionFromSignedData(chain, symbol, signedTx)
    if err != nil {
        return sdk.ErrInvalidTx(fmt.Sprintf("QueryUtxoTransactionFromSignedData Error:%v", signedTx)).Result(), txHash
    }

    if tx.From != fromAddr {
        return sdk.ErrInvalidTx(fmt.Sprintf("from an unexpected address:%v, expected address:%v", tx.From, fromAddr)).Result(), txHash
    }

    rawTx, _, err := keeper.cn.QueryAccountTransactionFromData(chain, symbol, rawData)
    if tx.To != rawTx.To {
        return sdk.ErrInvalidTx(fmt.Sprintf("to an unexpected address:%v, expected address:%v", tx.To, rawTx.To)).Result(), txHash
    }

    if !tx.Amount.Equal(rawTx.Amount) {
        return sdk.ErrInvalidTx(fmt.Sprintf("amount mismatch,expected:%v, actual:%v", rawTx.Amount, tx.Amount)).Result(), txHash
    }

    if !tx.GasPrice.Equal(rawTx.GasPrice) {
        return sdk.ErrInvalidTx(fmt.Sprintf("gasPrice mismatch, expected:%v, actual:%v", rawTx.GasPrice, tx.GasPrice)).Result(), txHash
    }

    if !tx.GasLimit.Equal(rawTx.GasLimit) {
        return sdk.ErrInvalidTx(fmt.Sprintf("gasLimit mismatch, expected:%v, actual:%v", rawTx.GasLimit, tx.GasLimit)).Result(), txHash
    }
    txHash = tx.Hash

    return sdk.Result{}, txHash
}
```

Listing 3.22: bhchain/x/transfer/keepers/utils.go

Recommendation Add necessary error handling in `verifyAccountBasedSignedTx`.

3.16 Blackhole Receipt Addresses of MsgSend/MsgMultiSend

- ID: PVE-016
- Severity: Informational
- Likelihood: Low
- Impact: Low
- Target: `CKB`
- Category: Coding Practices [16]
- CWE subcategory: CWE-684 [22]

Description

Besides those message types discussed in earlier sections, the `transfer` module also processes `MsgSend/MsgMultiSend` messages to transfer assets among addresses in HBTC Chain. However, the receipt addresses of these messages have not gone through rigorous and thorough validity checks. As a result, an HBTC Chain non-compliant (or illegitimate) receipt address may be given out of typo or typing mistakes and even the address format is not valid, the handler still proceeds to the end by sending the fund to the non-compliant address, resulting in unrecoverable funds as well as frustrated users.

```go
// ValidateBasic Implements Msg.
func (msg MsgSend) ValidateBasic() sdk.Error {
  if msg.FromAddress.Empty() {
    return sdk.ErrInvalidAddress("missing sender address")
  }

  if msg.ToAddress.Empty() {
    return sdk.ErrInvalidAddress("missing receipt address")
  }

  if !msg.Amount.IsValid() {
    return sdk.ErrInvalidCoins("send amount is invalid: " + msg.Amount.String())
  }
  if !msg.Amount.IsAllPositive() {
    return sdk.ErrInsufficientCoins("send amount must be positive")
  }
  return nil
}
```

Listing 3.23: bhchain/x/transfer/types/mesgs.go

Recommendation Apply additional checks to ensure the receipt addresses in both `MsgSend` and `MsgMultiSend` are compliant with the address formality in HBTC Chain.

```go
// ValidateBasic Implements Msg.
func (msg MsgSend) ValidateBasic() sdk.Error {
  if msg.FromAddress.Empty() || !msg.FromAddress.IsValidAddr()  {
    return sdk.ErrInvalidAddress("missing or wrong sender address")
  }
```

```
  if msg.ToAddress.Empty() || !msg.ToAddress.IsValidAddr() {
    return sdk.ErrInvalidAddress("missing or wrong receipt address")
  }

  if !msg.Amount.IsValid() {
    return sdk.ErrInvalidCoins("send amount is invalid: " + msg.Amount.String())
  }
  if !msg.Amount.IsAllPositive() {
    return sdk.ErrInsufficientCoins("send amount must be positive")
  }
  return nil
}
```

Listing 3.24: bhchain/x/transfer/types/mesgs.go (revised)

3.17 Unrecognized Contract-Sourced ETH Deposits in Chainnode

- ID: PVE-017
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `chainnode`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [18]

Description

HBTC Chain has a `chainnode` component that actively listens to (and synchronizes with) external chains such as `BTC` and `Ethereum`. This component is important in timely recognizing user deposits that eventually kick off subsequent collection operations.

We show below the code snippet that is part of `Ethereum` adaptor. In particular, it builds a response with incoming deposits of both `ETH` and other `ERC-20` assets. Notice that `ETH` deposits are recognized only for transaction destination (`toAddress.String()`) at the outer layer. In other words, other deposits from internal transactions are ignored and thus not accepted. The deposits of `ERC-20` assets are properly recognized by following the standard `ERC-20` `Transfer` event specification, regardless of internal-transaction deposits or straightforward EOA deposits.

As a result, current implementation may miss `ETH` deposits from internal transactions. Considering the growing popularity of smart wallets in cooperation with various DeFi protocols, it may need to be revisited to accommodate `ETH` deposits from these smart wallets (and these deposits are part of internal transactions).

```
    costFee := new(big.Int).Mul(new(big.Int).SetUint64(receipt.GasUsed), tx.GasPrice()
        )
```

```go

        if tx.Value().Cmp(big.NewInt(0)) == 1 {
            replyCh <- &proto.QueryAccountTransactionReply{
                TxHash:          tx.Hash().String(),
                TxStatus:        proto.TxStatus_Success,
                From:            sender.String(),
                To:              toAddress.String(),
                Amount:          tx.Value().String(),
                Memo:            "",
                Nonce:           tx.Nonce(),
                GasLimit:        new(big.Int).SetUint64(tx.Gas()).String(),
                GasPrice:        tx.GasPrice().String(),
                CostFee:         costFee.String(),
                BlockHeight:     uint64(height),
                BlockTime:       block.Time(),
                SignHash:        signer.Hash(tx).Bytes(),
                ContractAddress: "",
            }
        }
        for _, receiptLog := range receipt.Logs {
            if receiptLog.Removed {
                continue
            }
            if len(receiptLog.Topics) != 3 {
                continue
            }
            if receiptLog.Topics[0] != common.HexToHash("0
                xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef") {
                continue
            }

            tokenFromAddress := common.BytesToAddress(receiptLog.Topics[1].Bytes())
            tokenToAddress := common.BytesToAddress(receiptLog.Topics[2].Bytes())
            tokenAmount, ok := big.NewInt(0).SetString(fmt.Sprintf("%x", receiptLog.Data),
                16)
            if !ok {
                errCh <- errors.New("failed to decode token amount from receipt log data")
                needStop.Store(true)
                return
            }
            if tokenAmount.Cmp(big.NewInt(0)) == 1 {
                replyCh <- &proto.QueryAccountTransactionReply{
                    TxHash:          tx.Hash().String(),
                    TxStatus:        proto.TxStatus_Success,
                    From:            tokenFromAddress.String(),
                    To:              tokenToAddress.String(),
                    Amount:          tokenAmount.String(),
                    Memo:            "",
                    Nonce:           tx.Nonce(),
                    GasLimit:        new(big.Int).SetUint64(tx.Gas()).String(),
                    GasPrice:        tx.GasPrice().String(),
                    CostFee:         costFee.String(),
```

```
            BlockHeight:      uint64(height),
            BlockTime:        block.Time(),
            SignHash:         signer.Hash(tx).Bytes(),
            ContractAddress:  receiptLog.Address.String(),
          }
        }

      }
```

Listing 3.25: chainnode/chainadaptor/ethereum/ethereum.go

Meanwhile, to apply utmost precaution against so-called fake deposits, it is strongly suggested to perform balance difference check between current blockheight and the previous one (i.e., with `blockheight-1`). Any inconsistency warrants a manual follow-up for unambiguous resolution.

Recommendation For better DeFi adoption, it is recommended to accept deposits of `ETH` assets sourced from smart contracts.

3.18 Slashing Non-Cooperating Members in Key Management

- ID: PVE-016
- Severity: Informational
- Likelihood: Medium
- Impact: N/A
- Target: `keymanager`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [18]

Description

In HBTC Chain, the `settle` component is responsible for handling upcoming deposits as well as various requests for (distributed) key generation and signing. It also coordinates with active validators in a distributed and coordinated manner to generate and sign with individual secret shares. If a validator may become non-cooperative, it could cripple the entire scheme that has been adopted for fast and robust multi-party key generation and signing protocol [23].

We notice that the adopted protocol [23] assumes a threat model, i.e., `dishonest majority`, meaning that the number of players or validators the adversary corrupts, can be up to $n-1$ where n is the number of active validators in current epoch. In other words, the protocol does not guarantee that it will complete, despite the importance of maintaining `liveness` of the key generation and signing protocol. Without `liveness`, assets under custody may not be available, hence leading to serious denial-of-service consequences.

The mis-aligned threat models between multi-party key generation/signing and the `liveness` requirement of HBTC Chain makes it necessary to timely detect and punish non-cooperating parties. We notice various phases in both key generation and key signing could expose the wrong-doing actors. And it is imperative to develop and apply incentive schemes to ensure that validators will fully cooperate and maintain a healthy network for their duties by closely and actively participating on both key generation and signing. In the meantime, we may discern non-cooperative parties for unintended or intended wrong-doings. For unintended behaviors, they may be caused by configuration issues or temporary network partitions that caused the lack of participation of key generation and signing. For intended behaviors, they may be considered hostile and active punishment measures are deemed necessary.

Having said that, the exposure of wrong-doing validators is necessary and need to be directly linked together with the built-in `slashing` module. Currently, the desired linkage is still missing.

Recommendation Develop and apply necessary slashing mechanisms to reward cooperating members and disincentivize non-cooperating members in key management.

3.19 Proper Safe Prime Generation

- ID: PVE-019
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `dsign`
- Category: Coding Practices [16]
- CWE subcategory: N/A

Description

Upon receiving a request for key generation, the `settle` daemon bootstraps the decentralized key-generation protocol [23] to generate secret key shares for participating parties (i.e., validators). Assume that the protocol runs among `n` parties: `P1, ..., Pn`, and the parties run on input threshold `t` and chosen elliptic curve parameters. It typically has the following three rounds:

- `Commitment Round:` Each party i randomly generates a secret number u_i, and broadcasts a commit to the random point $Y_i = u_i * G$; Later on, each party broadcasts the corresponding decommitment to Y_i, so that each party can independently verify the correctness for $n - 1$ received decommitments. If there is any inconsistency, the protocol is aborted.

- `VSS Round:` Each party i participates in the (t, n) `Feldman-VSS` of the value u_i. The group public key is the resulting $Y = \Sigma_j Y_j$ and the local secret share of party i is $x_i = \Sigma_j f_j(i)$. Each party j randomly chooses the coefficients for the polynomial function $f_j(x)$ and privately sends the calculated $f_j(i)$ result to party i. Note that the coefficients essentially defines the polynomial and is the gist behind the Shamir secret-sharing algorithm (`sssa`). To properly notify other parties of its resulting secret share, the party i broadcasts a zero-knowledge proof of x_i (via discrete logarithms). Each party can independently verify other $n-1$ proofs and, if the proof fails, the protocol is aborted as well.

- `Paillier KeyGen Round:` Each party i generates a `Paillier keypair` and broadcasts the public key e_i. Behind the scheme, each party generates its own safe primes p_i and q_i required by the `Paillier keypair` and broadcasts their zero-knowledge proofs of p_i, q_i such that $N_i = p_i q_i$ (Note that N_i is the RSA modulus associated with the `Paillier` encryption e_i). Similarly, each party independently verifies $n-1$ received proofs and aborts otherwise.

It is important to note possible implications from square root attacks [24] that could affect the `Paillier KeyGen Round`. Specifically, the best algorithms to compute discrete logarithms in arbitrary groups (of prime order) are the `baby-step giant-step` method, the `rho` method and the `kangaroo` method. These methods differ in their complexity and memory-space tradeoffs. To avoid these attacks, a best practice is to ensure the prime numbers p_i, q_i behind the RSA modulus N have sufficiently large difference (typically 1024 bits).

However, it appears that the prime numbers generated in `RSAParameter()` do not follow the above best practice. Notice that it does have certain checks in place to ensure the generated `PTilde` and `QTilde` are not identical (line 30 in the code snippet below). However, it is also equally important to ensure their difference is sufficiently large to avoid the above mentioned square-root attacks.

```
15 func RSAParameter(bits int) (*big.Int, *big.Int, *big.Int, *big.Int, *big.Int, error) {
16   //gP^(PTilde-1)=gP^2p=1 mod PTilde
17   PTilde, gP, err1 := safePrimeAndGenerator(bits)
18   for err1 != nil {
19     fmt.Println("SafePrimeAndGenerator 1 fail!")
20     PTilde, gP, err1 = safePrimeAndGenerator(bits)
21   }
22   //gQ^(QTilde-1)=gQ^2q=1 mod QTilde
23   QTilde, gQ, err2 := safePrimeAndGenerator(bits)
24   for err2 != nil {
25     fmt.Println("SafePrimeAndGenerator 2 fail!")
26     QTilde, gQ, err2 = safePrimeAndGenerator(bits)
27   }
28
29   //Chinese Remainder Theorem requires gcd(m1,m2)=1
30   for PTilde.Cmp(QTilde) == 0 {
31     fmt.Println("Same safe prime!")
32     PTilde, gP, err1 = safePrimeAndGenerator(bits)
33     for err1 != nil {
34       fmt.Println("SafePrimeAndGenerator 1 fail!")
35       PTilde, gP, err1 = safePrimeAndGenerator(bits)
36     }
37     QTilde, gQ, err2 = safePrimeAndGenerator(bits)
38     for err2 != nil {
39       fmt.Println("SafePrimeAndGenerator 2 fail!")
40       QTilde, gQ, err2 = safePrimeAndGenerator(bits)
```

```
    }
  }
  p := big.NewInt(0).Rsh(big.NewInt(0).Sub(PTilde, one), 1)
  q := big.NewInt(0).Rsh(big.NewInt(0).Sub(QTilde, one), 1)

  NTilde := big.NewInt(0).Mul(PTilde, QTilde)
  t1 := big.NewInt(0).ModInverse(QTilde, PTilde)
  t2 := big.NewInt(0).ModInverse(PTilde, QTilde)
  b01 := big.NewInt(0).Mul(big.NewInt(0).Mul(gP, t1), QTilde)
  b02 := big.NewInt(0).Mul(big.NewInt(0).Mul(gQ, t2), PTilde)
  b0 := big.NewInt(0).Mod(big.NewInt(0).Add(b01, b02), NTilde)

  ...

  return NTilde, PTilde, QTilde, h1, h2, nil
}
```

Listing 3.26: dsign/primes/primes.go

Recommendation It is strongly recommended to ensure that safe primes generated are of the desired quality and length. In particular, when generating two RSA safe primes p_i and q_i for Paillier encryption with $N = p_i q_i$, there is a need to ensure that the difference $p_i - q_i$ is also very large (say 1020 bits) in order to avoid square-root attacks.

3.20 Unconstrained Private Key Range in sssa.Create()

- ID: PVE-020
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: sssa
- Category: Arg.s and Parameters [25]
- CWE subcategory: N/A

Description

HBTC Chain makes the unique innovation in creating, deploying, and managing secret shares among validators to enable cross-chain assets and their exchanges. The secret shares are developed based on the known Shamir secret sharing algorithm (sssa). The idea behind sssa is that it takes $k + 1$ points to define a polynomial of degree k. For example, 2 points defines a line, 3 points defines a parabola, 4 points defines a cubic curve and so forth.

For a (k, n) threshold scheme to share our secret S, the sssa algorithm typically chooses a random polynomial of degree $k - 1$ with free term the secret. The polynomial does not take zero coefficients. Also, the polynomial usually operates in a finite field F of size P where $0 < k <= n < P$; $S < P$ and P is a large prime number.

```
55 func keyGen(t, n int, coeff []*big.Int, privateKeyShare ...*btcec.PrivateKey) (
56   *btcec.PrivateKey, map[string]sssa.ShareXY, []*btcec.PublicKey) {
57   var newPriKey *btcec.PrivateKey
58   if len(privateKeyShare) > 0 {
59     newPriKey = privateKeyShare[0]
60   } else {
61     newPriKey, _ = btcec.NewPrivateKey(btcec.S256())
62   }
63   share, cof := sssa.Create(t, n, newPriKey.D, coeff)
64   return newPriKey, share, getCofCommits(cof)
65 }
```

Listing 3.27: dsign/dstsign/multisign.go

We emphasize that the secret key S for secret sharing needs to be smaller than the primer number P used for modulus operation, i.e., $S < P$. If we follow the key generation execution path, we notice that a private key may be dynamically generated (in the above `keyGen` function at line 61) and directly passed to the `sssa` for secret share generation. Within the `sssa` algorithm, there is no check applied to ensure $S < P$. The lack of $S < P$ could potentially corrupt the generation of secret shares and may lead to unrecoverable loss of secret keys.

```
/**
 * Returns a new arary of secret shares (encoding x,y pairs as base64 strings)
 * created by Shamir's Secret Sharing Algorithm requring a minimum number of
 * share to recreate, of length shares, from the input secret raw as a string
**/
func Create(minimum int, shares int, priKey *big.Int, coeff []*big.Int) (map[string]
    ShareXY, []*big.Int) {
  // Verify minimum isn't greater than shares; there is no way to recreate
  // the original polynomial in our current setup, therefore it doesn't make
  // sense to generate fewer shares than are needed to reconstruct the secret.

  // Convert the secret to its respective 256-bit big.Int representation
  //var secret []*big.Int = splitByteToInt([]byte(raw))
  copy := big.NewInt(0).Set(priKey)
  copy = copy.Mod(copy, prime)
  secret := big.NewInt(0).Set(copy)

  // List of currently used numbers in the polynomial
  var numbers []*big.Int = make([]*big.Int, 0)
  numbers = append(numbers, big.NewInt(0))
  var coefficients []*big.Int = make([]*big.Int, 0)
  ...
}
```

Listing 3.28: sssa-golang/sssa.go

Recommendation Apply the $S < P$ check for proper generation of Shamir secret shares.

3.21 Zeroizing Secret Temporary Values

- ID: PVE-021
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `KeyManager`
- Category: Coding Practices [16]
- CWE subcategory: CWE-1091 [26]

Description

In cryptography, the use of sensitive parameters (e.g., encryption private keys or passphrases) will usually leave undesirable memory footprints and these footprints should be better erased. Zeroization is the typical common practice of erasing these sensitive parameters (encryption private keys, passphrases, and critical security parameters) from a cryptographic module to prevent their disclosure.

We have examined a few routines whose computations centralize on these sensitive parameters and found that the above common practice needs to be applied. One example is the `handleBeginMsg` method of `keyGenHandler` in file `key_gen.go`. This method is part of key generation service and the local variable `keyShare` in line 247 contains its local secret share. After its use, it is recommended to erase it by assigning zero to it.

```
244
245 func (h *keyGenHandler) handleBeginMsg(ch []chan net.MultiStageObject) (*keyGenSession, error) {
246     msg := <-ch[keyGenStageBgein]
247     session := h.getKeyGenSession(msg.SessionKey())
248     keyShare, err := btcec.NewPrivateKey(btcec.S256())
249
250     if err != nil {
251         return session, err
252     }
253
254     var coeff = make([]*big.Int, len(session.keyNodes))
255     for i, v := range session.keyNodes {
256         coeff[i] = addressToLabel(v)
257     }
258     // var comm dstservice.Communicator
259     signHandler := &keySignHandler{}
260     signSession := &keySignSession{}
261     bhcoreComm := NewBHCoreCommunicator(ch, h, session, signHandler, signSession)
262     // comm = bhcoreComm
263
264     NTilde, PTilde, QTilde, h1, h2, err := primes.RSAParameter(RSALength)
265     if err != nil {
266         return session, err
267     }
268     trueShare := &dstsign.HonestShare{}
```

```go
    trueSchnorr := &dstsign.HonestSchnorr{}
    truePQProof := &dstsign.HonestPQProof{}

    labelBigInt := addressToLabel(h.km.b.GetBaseAddress())
    label := labelBigInt.String()

    _, tempNodeKey, _, _, _, err := dstsign.GetPublicKey(label, int(session.
        signThreshold),
        len(session.keyNodes), PQProofK, bhcoreComm, maxRand, coeff, NTilde, PTilde,
            QTilde, h1, h2,
        trueShare, trueSchnorr, truePQProof, keyShare)
    session.nodeKey = tempNodeKey
    if err == nil {
        session.nodeKey.KeyNodes = session.keyNodes
    }

    return session, err
}
```

Listing 3.29: settle/keymanager/key_gen.go

Similar issues can also be found in `settle/server/start.go#L133`, `bhchain/crypto/keys/hd/hdpath.go#L191`, `bhchain/crypto/keys/keybase.go#L174`, and `bhchain/crypto/keys/keybase.go#L346`.

We notice that the `settle` daemon supports the use of an environment variable to pass the sensitive password information (via `passphrase := os.Getenv("PASSWORD")`). While convenient, such use should be used only in debug/test environment, not production.

Recommendation Apply necessary zeroization of sensitive encryption keys and passphrases immediately after their uses.

3.22 Missing Validity Check in MtAwc

- ID: PVE-023
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `keymanager`
- Category: Security Features [19]
- CWE subcategory: CWE-285 [27]

Description

Following the previous section that examines the `keygen` protocol, we in this section analyze the `keysign` protocol. As mentioned earlier, the `keysign` protocol strictly follows the `multi-party ECDSA algorithm` [23].

This algorithm has five key inter-dependent phases: `commitment`, `MtA/MtAwc share conversion`, δ^{-1} `reconstruction`, r `generation`, and s `generation`. The first phase guarantees the non-repudiation of chosen random numbers that are extensively used in following phases. The second phase leverages an additively homomorphic scheme, i.e., `Paillier` encryption, to convert multiplicative shares of a secret to additive ones. This conversion is necessary to ensure the $t+1$ parties (instead of $2t+1$) are sufficient for the final signature generation. We notice that the protocol in this second phase requires the share conversion of two sets of random numbers (chosen from the first phase) and every pair of players P_i and P_j engages in two multiplicative-to-additive share conversion sub-protocols: `MtA` and `MtAwc`. The third phase reconstructs δ^{-1} that is needed for the four phase to compute r, the random number component of `ECDSA`. Finally, the five phase generates the signature component of `ECDSA`, i.e., s.

If we zoom in the second phase, there are two multiplicative-to-additive share conversions: `MtA` and `MtAwc`. The difference is `MtAwc` performs an extra check to ensure the participating party P_i use the correct secret share value, hence the name as `MtA with check`.

```go
func (t *Node) GetKeySignPhase2MsgSent(re Response) ([]SendingCheaterEvidence, error) {
  t.KeySignPhase2MsgSent = make([]types.KeySignPhase2Msg, t.P-1)
  errStr := ""
  var evidenceList []SendingCheaterEvidence = make([]SendingCheaterEvidence, 0)
  for k, v := range t.KeySignPhase1MsgReceived { //KeySignPhase1MsgReceived is not self-included
    if !t.CheckSenderRangeProof(v.GetNativeSenderRangeProofK(), v.MessageK, v.GetNativePaillierPubKey()) {
      errStr = errStr + v.LabelFrom + "K\n"
      temp := SendingCheaterEvidence{v.LabelFrom, v.GetNativeSenderRangeProofK(), v.MessageK, v.GetNativePaillierPubKey()}
      evidenceList = append(evidenceList, temp)
    }
    if !t.CheckSenderRangeProof(v.GetNativeSenderRangeProofR(), v.MessageR, v.GetNativePaillierPubKey()) {
      errStr = errStr + v.LabelFrom + "R\n"
      temp := SendingCheaterEvidence{v.LabelFrom, v.GetNativeSenderRangeProofR(), v.MessageR, v.GetNativePaillierPubKey()}
      evidenceList = append(evidenceList, temp)
    }
    if errStr != "" {
      continue
    }
    t.KeySignPhase2MsgSent[k].LabelFrom = t.label
    t.KeySignPhase2MsgSent[k].LabelTo = v.LabelFrom
    var Rk, Rr *big.Int
    nTilde, h1, h2 := t.NTilde[v.LabelFrom], t.h1[v.LabelFrom], t.h2[v.LabelFrom]
    pub := v.GetNativePaillierPubKey()
    oneCipher, oneR := PaillierEnc(big.NewInt(1), pub)
    t.KeySignPhase2MsgSent[k].MessageKResponse, Rk = getAnotherPart(v.MessageK, pub, t.randNumArray[k], t.r, oneCipher, oneR)
    t.KeySignPhase2MsgSent[k].MessageRResponse, Rr = getAnotherPart(v.MessageR, pub, t.randNumArray[k], t.prtKey, oneCipher, oneR)
    reR, rePrtKey := re.respond(t.r, t.prtKey)
    proofK := t.GetReceiverRangeProof(reR, t.randNumArray[k], Rk, v.MessageK, v.
```

```
        GetNativePaillierPubKey(), nTilde, h1, h2)
875     proofR := t.GetReceiverRangeProof(rePrtKey, t.randNumArray[k], Rr, v.MessageR, v.
        GetNativePaillierPubKey(), nTilde, h1, h2)
876     t.KeySignPhase2MsgSent[k].SetNativeReceiverRangeProofK(proofK)
877     t.KeySignPhase2MsgSent[k].SetNativeReceiverRangeProofR(proofR)
878   }
879   if errStr != "" {
880     return evidenceList, errors.New(errStr)
881   }
882   return nil, nil
883 }
```

Listing 3.30: settle/keymanger/multisign.go

The above code snippet shows the `GetKeySignPhase2MsgSent()` routine that prepares the message used in the second phase. The share conversions of `MtA` and `MtAwc` are processed in the `getAnotherPart()` subroutine (invoked twice in lines 871 and 872). We notice the extra check required in `MtAwc` is not performed. The lack of this extra check significantly weakens the security guarantee of the entire protocol as a player P_i may provide an incorrect key share to mislead the generation of signature without being detected.

```
func getAnotherPart(message []byte, pubKey *gaillier.PubKey, randomNum, ownNum *big.Int,
    oneCipher []byte, oneR *big.Int) ([]byte, *big.Int) {
  cA := message
  gama := randomNum
  b := ownNum
  pub := pubKey
  encGama := gaillier.Mul(pub, oneCipher, gama.Bytes())
  r := big.NewInt(0).Exp(oneR, gama, pub.Nsq)
  cB := gaillier.Mul(pubKey, cA, b.Bytes())
  cB = gaillier.Add(pubKey, cB, encGama)

  return cB, r
}
```

Listing 3.31: settle/keymanger/multisign.go

Recommendation Ensure `MtAwc` is indeed `MtAwc`, not `MtA`.

4 Conclusion

In this security audit, we have analyzed the HBTC Chain and related modules. During the first phase of our audit, we studied the source code and ran our in-house analyzing tools through the codebase. A list of potential issues were found, and some of them involve unusual interactions among multiple modules. And we have accordingly developed various test cases to reproduce and verify each of them. After further analysis and internal discussion, we determined that a number of issues need to be brought up and paid more attention to, which are reported in Sections 2 and 3.

Our impression through this audit journey is that the HBTC Chain is thoroughly designed and well engineered. The codebase is neatly organized and the modules are elegantly implemented. The identified issues are promptly confirmed and fixed. We'd like to commend HBTC Chain for a well-done software project, and for quickly fixing issues found during the audit process. Also, as expressed in Section 1.4, we appreciate any constructive feedback or suggestions about this report.

References

[1] PeckShield. PeckShield Inc. https://www.peckshield.com.

[2] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[3] Lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[4] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[5] MITRE. CWE-452: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[6] MITRE. CWE-459: Incomplete Cleanup. https://cwe.mitre.org/data/definitions/459.html.

[7] Tendermint Inc. Cosmos SDK Documentation. https://docs.cosmos.network/.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE-666: Operation on Resource in Wrong Phase of Lifetime. https://cwe.mitre.org/data/definitions/666.html.

[10] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

[11] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[12] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[13] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[14] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[15] MITRE. CWE: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[16] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[17] MITRE. CWE: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.html.

[18] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[19] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[20] MITRE. CWE-287: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

[21] MITRE. CWE-1068: Empty Code Block. https://cwe.mitre.org/data/definitions/1071.html.

[22] MITRE. CWE-684: Incorrect Provision of Specified Functionality. https://cwe.mitre.org/data/definitions/684.html.

[23] Rosario Gennaro and Steven Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless. In Proceedings of the 25th ACM Conference on Computer and Communications Security, CCS' 18, 2018.

[24] Edlyn Teske. Square-Root Algorithms For The Discrete Logarithm Problem (a Survey). In Public Key Cryptography and Computational Number Theory, 2001.

[25] MITRE. CWE CATEGORY: Often Misused: Arguments and Parameters. https://cwe.mitre.org/data/definitions/559.html.

[26] MITRE. CWE-1091: Use of Object without Invoking Destructor Method. https://cwe.mitre.org/data/definitions/1091.html.

[27] MITRE. CWE-285: Improper Authorization. https://cwe.mitre.org/data/definitions/285.html.