SOFTWARE AUDIT REPORT

for

BHOP CONSULTANTING PTE. LTD.

Prepared By: Shuxiao Wang

Hangzhou, China
Aug. 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(s)	Description
1.0	Aug. 15, 2020	Xuxian Jiang	Final Release Version
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

目录

1 介绍

我们（**PeckShield** [24]）受客户委托对 **HBTC Chain** 进行安全审计。根据我们的安全审计规范和客户需求，我们将在报告中列出用于检测潜在安全问题的系统性方法，并根据检测结果给出相应的建议或推荐以修复安全问题或提高安全性。 分析结果表明，HBTC Chain 当前分支中的代码在安全性和性能上仍然有改进的空间。本文档对审计结果作了分析和阐述。

1.1 关于HBTC Chain

HBTC Chain 提供了基于区块链的新一代的去中心化资产托管和清算技术。HBTC Chain 利用近期发展的各种技术（如门限密码学和区块链），进一步与基于社区的去中心化共识相结合，很好地解决了许多传统中心化数字资产平台所面临的各种安全和信任问题。 HBTC Chain 的基本信息如表 1.1 所示，其 Git 仓库和哈希值（被审核的分支）如表 1.2所示。

表 1.1: HBTC Chain的基本信息

条目	描述
发行方	BHOP Consultanting Pte. Ltd.
官方网址	https://chain.hbtc.com/
合约语言	Go
审计方法	白盒
审计完成时间	Aug. 15, 2020

1.2 关于PeckShield

PeckShield (派盾) 是面向全球的业内顶尖区块链安全团队，以提升区块链生态整体的安全性、隐私性以及可用性为己任，通过发布行业趋势报告、实时监测生态安全风险，负责任曝光0day漏洞，以及提供相关的安全解决方案和服务等方式帮助社区抵御新兴的安全威胁。可以通过下列联系方式联络我们： Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

表 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

1.3 方法

首先，我们使用了模糊测试方法寻找边缘案例，这些情况可能无法在常规的内部测试中被找到。当模糊测试工具检测出来问题之后，PeckShield 的安全研究人员会人工分析并确认这些问题的正确性。除此之外，我们采用白盒审计的方法人工校验并审计了 HBTC Chain的设计和源代码，排查一切潜在的安全隐患。在必要情况下，我们会设计独立的测试用例来复现相关漏洞。在接下来的章节中，我们将会详细介绍风险模型以及审计流程。

表 1.3: Vulnerability Severity Classification

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

1.3.1 风险模型

为了检测评估的标准化，我们根据 OWASP Risk Rating Methodology [23] 定义下列术语:

- 可能性 表示某个特定的漏洞被发现和利用的可能性;
- 影响力 度量了（利用该漏洞的）一次成功的攻击行动造成的损失;

- 危害性 显示该漏洞的危害的严重程度;

可能性和影响力各自被分为三个等级：高， 中和低。 危害性由可能性和影响力确定，分为四个等级：严重， 高危， 中危和低危。如表 1.3所示。

1.3.2 模糊测试

模糊测试是一种自动化的软件测试技术，通过某种策略生成输入文件并将其作为目标程序的输入，观察目标程序的执行是否有崩溃（或任何非预期结果)，从而发现软件漏洞。 在审计的第一阶段，我们使用模糊测试技术来发现可能的边缘案例或异常的模块间交互，这些问题可能是内部测试无法覆盖的。 作为暴露可能存在的漏洞的最有效的方法之一，近年来，模糊测试技术已经成为许多安全研究人员的首选。 目前，有很多模糊测试工具和配套软件，可以帮助安全人员进行模糊测试并更高效地发现漏洞。 根据 HBTC Chain的特点，我们采用 AFL [1] 作为模糊测试的主要工具。

AFL (American Fuzzy Lop) 是一种面向安全的模糊测试器，它采用一种新型的编译时指令和遗传算法来自动化地发现有效的测试用例，从而触发目标程序中未出现过的内部状态。 自问世以来，AFL 在业界受到越来越多的欢迎，并证明了它的有效性。它在许多大型软件项目中发现了不少重大的软件问题。AFL 模糊测试的基本流程如下：

- 生成编译时指令，并记录执行路径等信息;
- 构建一些输入文件并加入到输入队列中，根据不同的策略改变输入文件;
- 当执行某一输入文件时触发崩溃或超时时，该文件会被记录下来，以便后续分析;
- 循环完成上述过程。

在整个 AFL 测试过程中，我们将根据触发崩溃的输入文件作为依据并尝试复现它。 对于每个案例，我们将进一步分析其根本原因，并检查它是否确实为一个漏洞。 一旦一个案例被确定为 HBTC Chain的漏洞，我们将使用白盒审计进一步分析它。

1.3.3 白盒审计

在模糊测试之后，我们通过手动分析源代码继续进行白盒审计。在这个阶段，我们主要测试目标软件的内部结构、设计以及编码。我们还会重点验证程序的输入和输出流是否符合预期，以及检查设计和实现的不一致性以加强安全性。PeckShield 的审计人员首先会全面审查和理解源代码，然后创建特定的测试用例，将其作为输入并分析输出。诸如内部安全漏洞、非预期输出、崩溃或者当前路径出错等问题，都会被严格的二次检查

区块链是一种创建分布式数据库的方法，区块链的三大底层技术分别是密码学、去中心化和共识模型。区块链存在其独有的安全问题，基于对区块链总体设计的理解，我们在本次审计中将整个区块链拆分为以下几大方面，并对每个方面进行了相应的检查：

表 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

表 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

- 数据和状态存储。这与保存区块链数据的数据库和文件有关;
- 网络层中的 P2P 网络、共识和交易模型。将其放在一起的原因是共识和交易逻辑是与网络层紧密结合的;
- 虚拟机、账户模型和激励模型。这本质上是区块链的执行层和业务层，很多区块链的业务规范逻辑都在这里实现;
- 系统合约和服务。这些都是系统级的、区块链范围内的管理合约和服务;
- SDK 安全。这包括了额外的 SDK 模块和示例代码，供开发者社区的分发和使用;
- 其他。 包括了任何不属于上述层级的模块，如通用加密或其他第三方库、其他的软件项目中使用的优化方法、设计和编码一致性等问题。

基于上述分类，我们在表 1.4 和表 1.5 中展示了本报告中的审计项目的详细清单。为了更好地描述我们识别出的每个问题，我们还根据 CWE-699 [22] 对审核结果进行了分类。CWE 是一个社区开发的针对软件漏洞类型分类的列表，其目的是围绕软件开发中经常遇到的概念更好地对问题进行分类和组织。我们使用表 1.6 中的 CWE 类别来对我们的发现进行分类。

表 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 免责声明

请注意该审计报告并不保证能够发现 HBTC Chain 存在的一切安全问题，即评估结果并不能保证在未来不会发现新的安全问题。我们一向认为单次审计结果可能并不全面，因而推荐采取多个独立的审计和公开的漏洞奖赏计划相结合的方式来确保合约的安全性。最后必须要强调的是，审计结果仅针对 HBTC Chain公链的安全性，不构成任何投资建议。

2 检测结果

2.1 总结

如前所述，我们在第一阶段主要审计了 HBTC Chain的源代码（包括相关库函数），并且使用静态分析器扫描了代码库。 我们主要关注了以下几个组件：`bhchain`, `bhsettle` 和 `chainnode`.。这一阶段的主要目的是要识别已知的漏洞，并人工确认这些漏洞是否存在。 随后，我们会人工审查业务逻辑，检查可能存在的与业务相关的漏洞。

表 2.1: The Severity of Our Findings

严重性	发现个数	
严重	0	
高危	4	■■■■
中危	8	■■■■■■■■
低危	5	■■■■■
参考	5	■■■■■
总计	22	

目前为止，我们确实审计出了一些潜在的安全问题：有一些是由先前不会被考虑到的边缘情况导致的；还有一些可能会被用户非常规的交互所触发。 针对以上的每一种情况，我们都设计了相关的测试用例来复现并验证正确性。 经过了数次的分析和内部讨论之后，我们认为一些问题需要被开发人员所留意。 所有这些问题都根据它们的严重性被分类到表 2.1 中。具体细节我们将在第 3章中予以详细讨论。

2.2 主要发现

总体来讲，HBTC Chain 的设计和代码实现都是优良的。但是在解决被识别出来的漏洞之前（见表 2.2），仍有一定的提升空间。在本次审计中，我们共发现4个高危漏洞、8个中危漏洞、5个低危漏洞和5个参考项。

表 2.2: Key Audit Findings

编号	严重性	名称	状态
PVE-001	高危	不完整创世状态影响后续升级	已修复
PVE-002	中危	不正确的模块初始化顺序	已修复
PVE-003	中危	handleMsgKeyGen() 中无成本生成密钥	已修复
PVE-004	中危	handleMsgKeyGenFinish() 中不正确的费用返还	已修复
PVE-005	高危	OpcuAssetTransfer() 中资产被锁定 (Lockdown)	已修复
PVE-006	高危	OpcuAssetTransfer() 中的存款被非预期移	已修复
PVE-007	低危	优化OpcuAssetTransferWaitSign() 中的价格计算精度	已修复
PVE-008	低危	OpcuAssetTransferWaitSign() 中账户余额检查不够准确	已修复
PVE-009	参考	SysTransfer() 中的必要性检查	已修复
PVE-010	参考	针对 MsgSend/MsgMultiSend 生成有意义的事件 (Events)	已修复
PVE-011	低危	ConfirmedDeposit() 中容忍错误处理	已修复
PVE-012	低危	针对小额存款（dust deposits）不正确的 AssetCoins 减少策略	已修复
PVE-013	高危	handleMsgOrderRetry() 中可能的洪泛攻击 (Flooding Attack)	已修复
PVE-014	参考	handleMsgDeposit() 中的不合法订单移除	已确认
PVE-015	中危	在 SignedTx 验证中缺少错误处理 (error handling)	已修复
PVE-016	参考	MsgSend/MsgMultiSend 中的黑洞收款地址	已修复
PVE-017	中危	在 Chainnode 中未能识别从合约账户发起的 ETH 存款	已确认
PVE-018	参考	在密钥管理中移除不合作成员 (non-cooperating member)	已修复
PVE-019	中危	生成正确的素数	已修复
PVE-020	中危	在 sssa.Create() 中不受限的私钥范围	已修复
PVE-021	低危	使用 0 填充密钥的临时值	已修复
PVE-022	中危	在 MtAwc 中缺少有效性检查	已确认

3 检测结果详情

3.1 不完整创世状态影响后续升级

- ID: PVE-001
- 危害性: 高
- 可能性: 高
- 影响力: 中
- 目标: `cu, token, hrc20, order, ...`
- 类型: Initialization & Cleanup [8]
- CWE 子类: CWE-459 [9]

描述

Cosmos-SDK 是一个流行的模块化框架，可被用于构建针对特定应用的区块链，HBTC Chain 正是基于它实现的。Cosmos-SDK 可以通过提供可组合模块，使开发人员快速开发基于其的区块链。 HBTC Chain 利用了一些现有的模块（有一定程度的修改），并进一步开发了自己的用于数字资产托管和清算的模块。例如：cu 提供了资产管理的托管单元，token 列出了可用于交易或被托管的代币，mapping 支持跨链资产映射，keygen 提供了跨链资产的动态密钥生成服务，hrc20 在 HBTC Chain 上实现了类似 ERC20 代币发行和转让操作。

Cosmos-SDK 框架允许各个组件维护一个状态集，并且需要定义相关的方法来初始化、验证并且导出这个状态集。我们认为这些状态对于区块链的创世状态的导入和导出至关重要，因为在后续的升级中需要这些状态。在当前的 HBTC Chain 代码库中，有几个模块并没有完全的将创世所有相关状态正确地导入或导出。因此，这可能会导致升级失败。

以 custodianunit（即 cu）模块为例，我们在下面展示了当前 InitGenesis 和 ExportGenesis 方法的实现。顾名思义，只要涉及到状态的导入或导出，这两个方法就会被执行。ExportGenesis 方法同时导出了 params 和 cus，但 InitGenesis 方法只导入了 params，而没有导入 cus。换句话说，对于升级后恢复运行的 HBTC Chain 来说，之前运行中所创建的 cus 状态可能丢失。

```
// 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

同时，值得注意的是 ValidateGenesis 的验证也是不完整的，没有对在创世状态下保存的 cus 状态做验证。

```
// 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

类似的和创世相关的问题也存在于其他模块中，包括 token、hrc20、order、transfer 和 keygen。同样的，它们在 InitGenesis、ExportGenesis 和 ValidateGenesis 中的各个方法也需要进行相应的修改。

修复方法 在受影响模块中正确地导入和导出必要的创世状态。

3.2 不正确的模块初始化顺序

- ID: PVE-002
- 危害性: 中
- 可能性: 中
- 影响力: 中
- 目标: `keygen, supply, distr, ...`
- 类型: Initialization & Cleanup [8]
- CWE 子类: CWE-459 [9]

描述

HBTC Chain 中的各种可组合模块都有自己的内部依赖关系，这在初始化和拆卸 （teardown）过程中必须遵守。例如，模块 genutils 必须在 staking 后执行，这样代币池（pool）就可以被创世账户的代币正确初始化。此外，capability 模块在某些时候必须被首先初始化，以便它可以初始化任何能力（capability）。基于此，其他模块就可以创建或声明（claim）某种已经被初始化并创建的能力。同样地，gov 和 slashing 也依赖于 bank 来访问或修改余额。

```
234     // During begin block slashing happens after distr.BeginBlocker so that
235     // there is nothing left over in the validator fee pool, so as to keep the
236     // CanWithdrawInvariant invariant.
237     app.mm.SetOrderBeginBlockers(mint.ModuleName, distr.ModuleName, slashing.ModuleName)
238     app.mm.SetOrderEndBlockers(crisis.ModuleName, gov.ModuleName, staking.ModuleName)
239
240     // NOTE: The genutils moodule must occur after staking so that pools are
241     // properly initialized with tokens from genesis accounts.
242     app.mm.SetOrderInitGenesis(
243         genaccounts.ModuleName, otypes.ModuleName, receipt.ModuleName, token.ModuleName,
                keygen.ModuleName, distr.ModuleName, staking.ModuleName,
244         custodianunit.ModuleName, transfer.ModuleName, slashing.ModuleName, gov.
                ModuleName,
245         mint.ModuleName, supply.ModuleName, crisis.ModuleName, genutil.ModuleName, hrc20
                .ModuleName, mapping.ModuleName,
246     )
```

Listing 3.3: bhchain/bhexapp/app.go

对当前代码库的检查之后，我们发现模块的初始化不一致（如上所示）。特别是 app.mm.SetOrderInitGenesis() 表示 InitGenesis 的顺序必须满足固有的依赖性。在图 3.1 中，我们整理了 HBTC Chain 中当前模块之间的实际依赖关系（通过检查每个模块中相关的模块间保持器（keeper）的引用关系得出的）。也就是说，如果 keygen 引用了 order，我们可以推断 keygen 依赖于 order，因此在依赖关系图中有 order -> keygen 这一关系。

通过比对 app.mm.SetOrderInitGenesis 中的依赖关系与上述的实际模块依赖图，我们发现了以下可能违反了模块依赖的情况：keygen、distr、staking、cu、transfer、gov、mint 和 supply。不恰当的顺序可能会导致初始化中断或引入错误的运行时状态，因此不恰当的依赖关系一定要被避免。

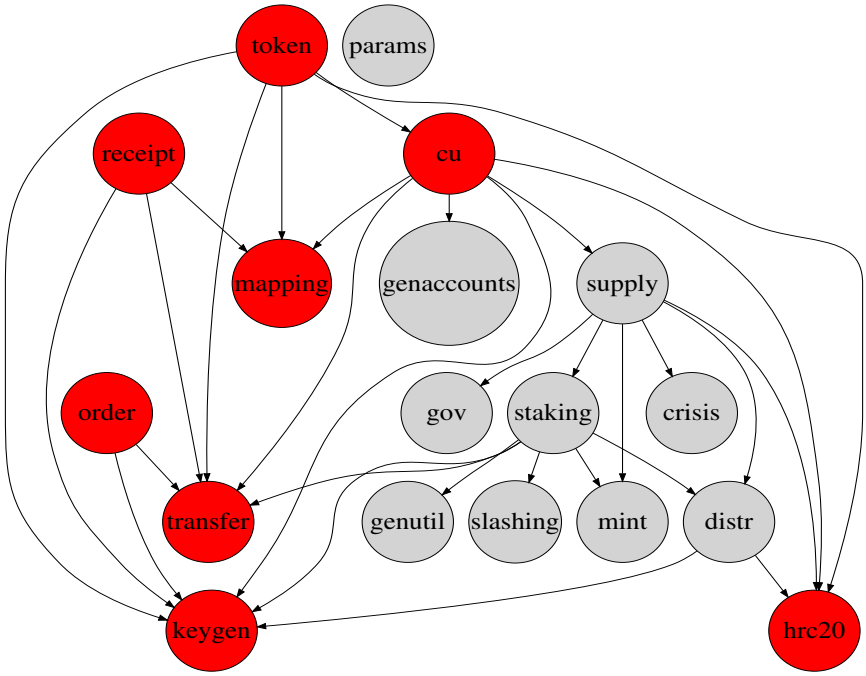

图 3.1: The Module Dependency in HBTC Chain

修复方法 相关的修复方法非常直接，我们需要根据模块间的内部实际依赖顺序来初始化所有模块。

```
241     // NOTE: The genutils moodule must occur after staking so that pools are
242     // properly initialized with tokens from genesis accounts.
243     app.mm.SetOrderInitGenesis(
245     /* Old Order
246         genaccounts.ModuleName, otypes.ModuleName, receipt.ModuleName, token.ModuleName,
                keygen.ModuleName, distr.ModuleName, staking.ModuleName,
247         custodianunit.ModuleName, transfer.ModuleName, slashing.ModuleName, gov.
                ModuleName,
248         mint.ModuleName, supply.ModuleName, crisis.ModuleName, genutil.ModuleName, hrc20
                .ModuleName,
249         mapping.ModuleName,
250     */
252         // New Order
253         otypes.ModuleName, receipt.ModuleName, token.ModuleName,
254         custodianunit.ModuleName, genaccounts.ModuleName, supply.ModuleName,
255         gov.ModuleName, staking.ModuleName, crisis.ModuleName, slashing.ModuleName,
256         genutil.ModuleName, mint.ModuleName, distr.ModuleName,
257         transfer.ModuleName, keygen.ModuleName, hrc20.ModuleName,
258         mapping.ModuleName,
```

```
259     )
```

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

3.3 handleMsgKeyGen() 中无成本生成密钥

- ID: PVE-003
- 危害性: 中
- 可能性: 中
- 影响力: 中
- 目标: keygen
- 类型: Business Logic [17]
- CWE 子类: CWE-666 [11]

描述

在 HBTC Chain 的所有模块中，kengen 模块是相对重要的。这个模块为跨链资产提供了动态密钥生成服务。具体来说，当用户请求在支持的外链中创建托管地址时（通过 MsgKeyGen 消息），keygen 会在 handleMsgKeyGen 句柄（handler）中处理该请求，这实质上是将该请求委托给 settle 守护进程（daemon）。

该模块通过 handleMsgKeyGen 句柄处理 MsgKeyGen 消息时区分了三种不同的场景：subToken、WaitAssignKeyGenOrder 和 KeyGenOrder。第一种 subToken 场景表示请求一个类 ERC20 资产的地址；第二种 WaitAssignKeyGenOrder 场景检查了是否存在一个预先生成的地址，如果存在的话则直接分配一个旧地址来响应请求；第三种 KeyGenOrder 场景将托管地址密钥生成的任务交给了 settle。

第二种场景中有一个问题允许了无限制生成密钥。具体来说，相关的开户费 feeCoins 并没有从发起请求的用户账户（fromCU）中扣除，但是这笔金额已经记入了 CommunityPool（如下面代码片段中第 158 行的 keeper.dk.AddToFeePool() 所示）。

```
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)
156
157         if feeCoins.IsAllGT(sdk.NewCoins(sdk.NewCoin(sdk.NativeToken, sdk.ZeroInt()))) {
158             keeper.dk.AddToFeePool(ctx, sdk.NewDecCoins(feeCoins))
159         }
160     ...
```

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

修复方法 在第二种场景下，正确地从请求账户（fromCU）中扣掉相应的开户费用即可。

3.4 handleMsgKeyGenFinish() 中不正确的费用返还

- ID: PVE-004
- 危害性: 中
- 可能性: 中
- 影响力: 低
- 目标: `dbswap.cpp`
- 类型: Coding Practices [16]
- CWE 子类: CWE-628 [10]

描述

正如第 3.1 节所述，keygen 模块为跨链资产提供动态密钥生成服务。具体来说，它有几个句柄来处理相应类型的消息。在前面的章节中，我们关注了处理 MsgKeyGen 消息的 handleMsgKeyGen() 句柄。在本节中，我们将研究另一个句柄： handleMsgKeyGenFinish()，被用于处理 MsgKeyGenFinish 消息。顾名思义，此消息通知 HBTC Chain 之前的密钥生成请求已经完成。

而 handleMsgKeyGenFinish() 内部存在一个问题：它不仅将开户费用返还给发起请求的用户，还将相同数量的开户费用转给了 distr 模块，这无意中导致了对 HBTC Chain 的原生代币 hbc 的非预期通胀。

具体来说，我们在下面展示了 handleMsgKeyGenFinish() 句柄内部的相关代码片段。在对 MsgKeyGenFinish 消息进行必要的检查后，系统最终通过将开户费用（之前在 fromCU.SubCoinsHold 上暂存）转移到 distr 中来收取相应费用。但第 305 行显示开户费用最终会被退回到 fromCU 中。

```
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 {
304         fromCU.SubCoinsHold(openFee)
305         fromCU.AddCoins(openFee)
306         keeper.ck.SetCU(ctx, fromCU)
307         keeper.dk.AddToFeePool(ctx, sdk.NewDecCoins(openFee))
308     }
309     ...
```

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

修复方法 不需要将用于生成密钥的开户费用返还给 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 {
304         fromCU.SubCoinsHold(openFee)
305         // fromCU.AddCoins(openFee)
306         keeper.ck.SetCU(ctx, fromCU)
307         keeper.dk.AddToFeePool(ctx, sdk.NewDecCoins(openFee))
308     }
309     ...
```

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

3.5 OpcuAssetTransfer() 中资产被锁定（Lockdown）

- ID: PVE-005
- 危害性: 高
- 可能性: 高
- 影响力: 高
- 目标: `transfer`
- 类型: Time and State [15]
- CWE 子类: CWE-362 [7]

描述

在 HBTC Chain 的所有模块中，transfer 模块是最关键的模块之一，其主要功能是转移 HBTC Chain 内外的资产。它的复杂程度也部分体现在它所可以识别和处理的消息数量上：总共有 21 种不同的消息类型，包括 MsgSend、MsgDeposit、MsgWithdraw、MsgSysTransfer、MsgOpcu AssetTransfer 及它们的变种。

在本节中，我们主要关注与 opcu 资产转移相关的四种消息类型： MsgOpcuAssetTransfer、MsgOpcuAssetTransferWaitSign、MsgOpcuAssetTransferSign Finish 和 MsgOpcuAssetTransferFinish。在这四种消息类型中，第一种消息类型（MsgOpcuAssetTransfer）旨在发起托管在 opcu 中的资产转移；第二种消息类型（MsgOpcuAssetTransferWaitSign）负责代表 opcu 启动密钥签署，这个密钥将会在验证者之间共享；第三种消息类型（MsgOpcuAssetTransferSignFinish）标志着密钥签名的完成，以便被该交易在链上进行广播和打包；第四种消息类型（MsgOpcuAsset TransferFinish）标志着成功完成转账以进行后续的状态更新和后续操作。

如果我们深入研究处理 MsgOpcuAssetTransfer 的逻辑，它的句柄 OpcuAssetTransfer 需要几个参数来指定相关的 opcu 账户 opCUAddr：新的目的地址 toAddr，资产符号 symbol，以及这笔转账中相关的 TransferItems。为了方便组织和管理整个转账流程，HBTC Chain 有其内部的 order 系统。而每一笔转账都有其唯一的 orderID。不同的转账会有不同的 orderID。

然而，这个句柄可能被不正确的使用，最终使转账资金被锁定。具体来说，它没有验证给定的目标地址 toAddr 的有效性（freshness）以确保它是在当前的迁移（migration） epoch 中生成的。因此，可以提供一个过时的（outdated）toAddr 来绕过以下检查（第 60 行）。

```
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      }

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

在校验之后，句柄进一步分辨出两种不同的代币类型：Utxo 和 account。第一种类型涉及到了 BTC 中的资产，第二种类型涉及到了 Ethereum 中的资产。对于 BTC 资产，每个 TransferItem 将相应地被标记为 DepositItemStatusInProcess，因此在转账完成之前资产可能不会被释放；对于 Ethereum 资产，opcu 账户将将被标记为 opCU.SetEnableSendTx(false，chain，fromAddr)，这将防止任何在其下托管的资产被转移，直到 flag 被设置为 true。

```
88          ...
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          for _, item := range items {
99              keeper.ck.SetDepositStatus(ctx, symbol, opCUAddr, item.Hash, item.Index, sdk
                    .DepositItemStatusInProcess)
100         }
101         ...
```

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

如果如上所示的 opcu 资产被转账，那么这笔资产将被转移到旧的 toAddr，将没有人持有共享密钥；如果转账过程没有完成，其内部状态也将被修改，这样做的目的有二：对于 BTC 资产，这样做可以防止相同的 TransferItems 被二次使用；对于 Ethereum 资产，这样做可以避免在 opCU 中托管的资产被转移。无论两者中任意情况发生，资金都会被锁定且无法被使用。

修复方法 增加额外的校验来确保 toAddr 的有效性，即它是在当前的 epoch 中生成的。

3.6 OpcuAssetTransfer() 中的存款被非预期移除

- ID: PVE-006
- 危害性: 高
- 可能性: 高
- 影响力: 高
- 目标: `transfer`
- 类型: Time and State [15]
- CWE 子类: CWE-362 [7]

描述

如前所述，transfer 模块是最关键的模块之一，其主要功能是允许资产在 HBTC Chain 内外进行转移。在上一节中，我们已经研究了一个与 MsgOpcuAssetTransfer 处理相关的漏洞。在本节中，我们将研究同一句柄中的另一个问题，该问题可能导致合法存款被意外的删除。

具体来说，OpcuAssetTransfer 句柄需要一些参数来指定相关的 Opcu 账户 opCUAddr：新的目的地址 toAddr，资产符号 symbol，以及这笔转账中相关的 TransferItems。为了方便组织和管理整个转账过程，HBTC Chain 有其内部的 order 系统。而每一笔转账都有其唯一的 orderID，不同的转账也会有不同的 orderID。

最后一个问题是关于参数 toAddr 的非有效性（non-freshness）的。这个问题与另一个参数 TransferItems 的项目重复有关。我们将在下面展示消息类型 MsgOpcuAssetTransfer 的有效性检查方法：ValidateBasic()。显然，TransferItems 上有一个检查以确保 TransferItems 数组不为空。换句话说，它不会检查数组中的项目是否重复。因此，我们可以在 TransferItems 中构造一个消息类型，其中包含有重复的项目。

```
1093 // quick validity check
1094 func (msg MsgOpcuAssetTransfer) ValidateBasic() sdk.Error {
1095     // note that unmarshaling from bech32 ensures either empty or valid
1096     _, err := sdk.CUAddressFromBase58(msg.FromCU)
1097     if err != nil {
1098         return ErrBadAddress(DefaultCodespace)
1099     }
1101     _, err = sdk.CUAddressFromBase58(msg.OpCU)
1102     if err != nil {
1103         return ErrBadAddress(DefaultCodespace)
1104     }
1106     if msg.ToAddr == "" {
1107         return ErrBadAddress(DefaultCodespace)
1108     }
1110     if msg.OrderID == "" {
1111         return ErrNilOrderID(DefaultCodespace)
1112     }
1114     if len(msg.TransferItems) == 0 {
1115         return sdk.ErrInvalidTx("transfer items are empty")
1116     }
1118     return nil
1119 }
```

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

此外，假设有一个 TransferItem A，它的金额小于 OpcuAstTransferThreshold。(如果没有，我们可以随意创建一个）。同时，还有另一个 TransferItem B，但其金额高于OpcuAstTransferThreshold。为简便起见，我们假设 A 的金额等于 0.1*OpcuAstTransferThreshold，B 的金额等于 1.1*OpcuAstTransferThreshold。因此，我们可以构造一个有两个 A 和一个 B 的数组，使得它们的总和满足第 98 行的条件检查，即 sum.LTE(keeper.utxoOpcuAstTransferThreshold(len(items), tokenInfo))。而右端的值是随着 TransferItems 中的项目数线性增长的，这意味着像这样构建的 TransferItems 总是合法有效的。

```
88          ...
89          sum := sdk.ZeroInt()
90          for _, item := range items {
91              depositItem := keeper.ck.GetDeposit(ctx, symbol, opCUAddr, item.Hash, item.
                    Index)
92              if depositItem == sdk.DepositNil  !depositItem.Amount.Equal(item.Amount)
93                  depositItem.Status == sdk.DepositItemStatusInProcess {
94                  return sdk.ErrInvalidTx(fmt.Sprintf("Invalid DepositItem(%v)", item.Hash
                        )).Result()
95              }
96              sum = sum.Add(item.Amount)
97          }

99          if sum.LTE(keeper.utxoOpcuAstTransferThreshold(len(items), tokenInfo)) {
100             for _, item := range items {
101                 keeper.ck.DelDeposit(ctx, symbol, opCUAddr, item.Hash, item.Index)
102             }
103             burnedCoins := sdk.NewCoins(sdk.NewCoin(symbol, sum))
104             opCU.AddGasUsed(burnedCoins)
105             keeper.ck.SetCU(ctx, opCU)
106             if keeper.checkUtxoOpcuAstTransferFinish(ctx, fromAddr, symbol, opCU) {
107                 opCU.SetMigrationStatus(sdk.MigrationFinish)
108                 keeper.ck.SetCU(ctx, opCU)
109                 keeper.checkOpcusMigrationStatus(ctx, curEpoch)
110             }
111             return sdk.Result{}
112         }
113         ...
```

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

一旦条件被满足，与 B 相关的存款项就会被认为过小而被"安全"的删除，最终造成 opcu 保管的资产出现损失。此外，它还会搞乱内部的 GasUsed 状态。

修复方法 增加额外的检查来确保 TransferItems 的唯一性。

3.7 优化OpcuAssetTransferWaitSign() 中的价格计算精度

- ID: PVE-007
- 危害性: 低
- 可能性: 中
- 影响力: 低
- 目标: `transfer`
- 类型: Numeric Errors [18]
- CWE 子类: CWE-190 [4]

描述

正如 3.5 节所述，Opcu 资产转移要求针对四种消息类型的处理：MsgSend、MsgDeposit、MsgWithdraw、MsgSysTransfer 和 MsgOpcuAssetTransfer。第一种消息类型（MsgOpcuAssetTransfer）

旨在发起托管在 opcu 中的资产转移；第二种消息类型（MsgOpcuAssetTransferWaitSign）负责代表 opcu 启动密钥签署，这个密钥将会在验证者之间共享；第三种消息类型（MsgOpcuAsset TransferSignFinish）标志着密钥签名的完成，以便被该交易在链上进行广播和打包；第四种消息类型（MsgOpcuAssetTransferFinish）标志着成功完成转账以进行后续的状态更新和后续操作。

当处理第二种消息类型时（MsgOpcuAsset TransferWaitSign），需要计算交易价格。然而，目前的计算方式会导致精度损失（第 246 行所示）：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)
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

为了有更高的精度，我们建议先计算乘法再计算除法，即：sdk.NewDecFromInt(tx.CostFee).MulInt64(sdk.KiloBytes).Quo(size)。

还有一个相似的问题在 CollectWaitSign 方法处理 MsgCollectWaitSign 消息时（第 104 行），也可以改进计算过程来得到更高的计算精度。

修复方法 修改原有的计算方法为 sdk.NewDecFromInt(tx.CostFee).MulInt64(sdk.KiloBytes).Quo(size) 来得到更低的精度损失。

3.8 OpcuAssetTransferWaitSign() 中账户余额检查不够准确

- ID: PVE-008
- 危害性: 低
- 可能性: 中
- 影响力: 低
- 目标: `transfer`
- 类型: Business Logic [17]
- CWE 子类: CWE-837 [20]

描述

在这一节中，我们检查了 3.7 节中曾检查过的 MsgOpcuAssetTransferWaitSign 处理逻辑并发现

了另一个逻辑问题。

具体来说，在启动密钥签名过程之前，我们需要确保发送方有足够的资金用于支付交易金额，以及相关的手续费。在下面的代码片段中，我们重点介绍了相关变量及其计算方法。很显然，当 order.Symbol == chain（第 289 行）时，手续费已经计入了 tx.Amount 中。后续增加的 coins = coins.Add(feeCoins) 则将手续费计算了两次：tx.GasPrice.Mul(tokenInfo.GasLimit) 和 tx.CostFee 中各一次。

```
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

修复方法 改正上述的有关手续费的计算流程。

3.9 SysTransfer() 中的必要性检查

- ID: PVE-009
- 危害性: 参考
- 可能性: 中
- 影响力: 未知
- 目标: `transfer`
- 类型: Business Logic [17]
- CWE 子类: CWE-837 [20]

描述

在本节中，我们研究了一个 SysTransfer() 函数针对 MsgSysTransfer 消息类型的处理逻辑。它用于收取必要的手续费当外部用户存款或内部 opcu 转账。

具体来说，这个 SysTransfer() 句柄需要以下几个参数来指定资金来源 fromCUAddr：目的地 toCUAddr 和它的外部地址 toAddr，资产符号 symbol，以及这笔转账的手续费 amt。同样地，为了便于组织和管理整个转账流程，也为这个特定的 SysTransfer 指定了唯一的 orderID，不同的转账将有不同的 orderID。

```
38      ...
39      toCU := keeper.ck.GetCU(ctx, toCUAddr)
40      if toCU == nil {
41          return sdk.ErrInvalidAccount(toCUAddr.String()).Result()
42      }
43      valid, canonicalToAddr := keeper.cn.ValidAddress(chain, symbol, toAddr)
44      if !valid {
45          return sdk.ErrInvalidAddr(fmt.Sprintf("%v is not a valid address", toAddr)).
                Result()
46      }
47      toCUAsset := toCU.GetAssetByAddr(symbol, canonicalToAddr)
48      if toCUAsset == sdk.NilAsset {
49          return sdk.ErrInvalidTx(fmt.Sprintf("%v does not belong to cu %v", toAddr, toCU.
                GetAddress().String())).Result()
50      }
51      if toCU.GetCUType() == sdk.CUTypeOp && toCU.GetMigrationStatus() == sdk.
            MigrationFinish {
52          if toCUAsset.Epoch != keeper.sk.GetCurrentEpoch(ctx).Index {
53              return sdk.ErrInvalidTx("Cannot sys transfer to last epoch addr").Result()
54          }
55      }

57      if keeper.hasProcessingSysTransfer(ctx, toCUAddr, chain, canonicalToAddr) {
58          return sdk.ErrInvalidTx(fmt.Sprintf("To OPCU %v has processing sys transfer of %
                s", toCUAddr, chain)).Result()
59      }
60      ...
```

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

在初始化 systransfer 之前，必须确保 toAddr 确实需要手续费以便将来的收款或转帐。如果它没有持有任何有意义的资产，就完全没有必要初始化这个程序。我们意识到可以使用 offline 方法来确保这一点。即使手续费金额可能并不明显，但是避免不必要的手续费浪费在 HBTC Chain 的编码逻辑中还是大有益处的。

修复方法 增加额外的检查确保 toAddr 确实需要手续费。

3.10 针对 MsgSend/MsgMultiSend 生成有意义的事件（Events）

- ID: PVE-010
- 危害性: 参考
- 可能性: 中
- 影响力: 未知
- 目标: `transfer`
- 类型: Coding Practices [16]
- CWE 子类: CWE-1116 [21]

描述

事件（event）和日志（log）是区块链的重要组成部分，它们可以极大地促进区块链活动

（activity）的封装和规范化信息传达，并将其暴露给外部监听的 DApp。为此，精确且规范化地生成事件或日志总是有益处的。

在分析 transfer 模块（它负责处理 21 种消息类型）的过程中，我们注意到该模块处理的两种消息类型并没有产生有意义的事件。具体来说，这两种消息类型（MsgSend 和 MsgMultiSend）产生的事件只是被封装在了通用事件类型 sdk.EventTypeMessage 中。它可能并不能精准的传达信息，因此强烈建议让它们对应的事件类型更加具体且有意义。

```
92  // Handle MsgSend.
93  func handleMsgSend(ctx sdk.Context, k keeper.Keeper, msg types.MsgSend) sdk.Result {
94    if !k.GetSendEnabled(ctx) {
95      return types.ErrSendDisabled(k.Codespace()).Result()
96    }

98    if k.BlacklistedAddr(msg.ToAddress) {
99      return sdk.ErrUnauthorized(fmt.Sprintf("%s is not allowed to receive transactions",
            msg.ToAddress)).Result()
100   }

102   result, err := k.SendCoins(ctx, msg.FromAddress, msg.ToAddress, msg.Amount)
103   if err != nil {
104     return err.Result()
105   }

107   ctx.EventManager().EmitEvent(
108     sdk.NewEvent(
109       sdk.EventTypeMessage,
110       sdk.NewAttribute(sdk.AttributeKeyModule, types.AttributeValueCategory),
111     ),
112   )

114   result.Events = append(result.Events, ctx.EventManager().Events()...)
115   return result
116 }
```

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

修复方法 在处理 MsgSend 和 MsgMultiSend 产生更具体的、更有意义的事件类型。

3.11 ConfirmedDeposit() 中容忍错误处理

- ID: PVE-011
- 危害性: 低
- 可能性: 低
- 影响力: 低
- 目标: `transfer`
- 类型: Business Logics [17]
- CWE 子类: CWE-841 [13]

描述

在本节中，我们重新审查了 transfer 模块，特别是检查了处理 MsgConfirmedDeposit 消息的逻辑。对于每个用户的存款请求，需要经过当前的大多数验证者的确认。而每个消息可以携带两个数组，一个是有效存款，另一个是无效存款。换句话说，一条消息可以同时批量处理多个 orderID。

为了详细说明，我们在下面展示了一个核心 helper 方法（processDepositOrderIDs）的代码片段。在这个方法里面，有一个 for 循环用来遍历每一个 orderID，并检查它是否已经被确认。确认的 orderID 将被返回给调用者。然而，我们发现如果在处理一个已被确认的 orderID 时出现了错误，那么其余的 orderID 将会被跳过。这些被忽略的 orderID 很可能不得不经过另一轮的消息提交和处理，从而延迟整个存款的过程。为了避免不必要的延迟，建议继续处理其余的 orderID，仅忽略导致错误的 orderID。

修复方法 修复方法非常直接，即在循环过程中不使用 return 而是 continue 语句。

3.12 针对小额存款（dust deposits）不正确的 AssetCoins 减少策略

- ID: PVE-012
- 危害性: 低
- 可能性: 中
- 影响力: 低
- 目标: `transfer`
- 类型: Business Logic [17]
- CWE 子类: CWE-841 [13]

描述

当启动在 opcu 托管下的资产转移时，需要验证被转账金额是否达到了特定的阈值。对于基于 Utxo 的代币类型，阈值由 utxoOpcuAstTransferThreshold 定义；对于基于账户的代币类型，阈值则由 tokenInfo.SysTransferAmount 定义。其目的是检测是否有必要转移如此小额的资产。当检测到这样的小额转账时，HBTC Chain 简单地认为它被销毁了，并从所拥有的 AssetCoins 中减少相同数量的金额。

我们的分析表明，对于基于账户的代币类型来说，情况确实如此。然而，对于基于 Utxo 的代币类型，小额转账已经被销毁，但并没有在被拥有的 AssetCoins 中减少（如下面代码片段中的第 103 行所示）。

```
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      }

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 3.16: bhchain/x/transfer/keeper/opcuasset_transfer.go

修复方法 正确减少对于小额存款的 AssetCoins

3.13 handleMsgOrderRetry() 中可能的洪泛攻击（Flooding Attack）

- ID: PVE-013
- 危害性: 高
- 可能性: 高
- 影响力: 中
- 目标: `transfer`
- 类型: Security Features [14]
- CWE 子类: CWE-284 [6]

描述

如前文所述，transfer 模块处理 21 种不同的消息类型。在本节中，我们重点介绍一种特殊消息类型（MsgOrderRetry）的处理逻辑。它的唯一目的是在之前的尝试没有结果的情况下重新提交订单。

这个消息类型有一个名为 RetryTimes 的字段。我们的分析表明，这个字段没有经过严格的检查。正因为如此，可以发送一系列 MsgOrderRetry 消息，这些消息具有相同的 OrderID

和 from，但具有不同的 RetryTimes。结果是当句柄继续执行到一个叫做 addOrderRetryConfirmNode 的函数时（见下面的代码片段），该函数将不断地把这些已确认的消息存储到本地中。而存储密钥是通过 retryOrderKey(strings.Join(orderIDs, "&"), retrytimes) 来计算的，这说明密钥长度可以由用户输入的 OrderID 来控制，这些被占用的存储空间将永远不会被释放，这最起码会造成资源浪费或低效。而当累计的垃圾数据占据了全部存储空间时，最终会危害到全链的各种正常操作。

```
264 func (keeper BaseKeeper) addOrderRetryConfirmNode(ctx sdk.Context, txID, validatorAddr
        string, retrytimes uint32, valsNum int) bool {
265   retryOrderConfirmNodes := []string{}
266   store := ctx.KVStore(keeper.storeKey)
267   bz := store.Get(retryOrderKey(txID, retrytimes))
268   if bz != nil {
269     keeper.cdc.MustUnmarshalBinaryBare(bz, &retryOrderConfirmNodes)
270   }

272   bFind := false
273   for _, v := range retryOrderConfirmNodes {
274     if v == validatorAddr {
275       bFind = true
276       break
277     }
278   }

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

285     //have been confirmed
286     if len(retryOrderConfirmNodes)-1 >= sdk.Majority23(valsNum) {
287       return false
288     }

290     if len(retryOrderConfirmNodes) >= sdk.Majority23(valsNum) {
291       return true
292     }
293   }

295   return false
296 }
```

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

修复方法 在 MsgOrderRetry 中增加对 RetryTimes 的检查。

3.14 handleMsgDeposit() 中的不合法订单移除

- ID: PVE-014
- 危害性: 参考
- 可能性: 高
- 影响力: 未知
- 目标: `transfer`
- 类型: Coding Practices [16]
- CWE 子类: CWE-1116 [21]

描述

在本节中，我们重点介绍另一种特殊的消息类型（MsgDeposit）的处理逻辑。它的主要目的是记录用户存款操作，并在需要的时候启动内部收款流程。

处理逻辑在函数 handleMsgDeposit() 中实现。在该函数中，我们注意到 MsgDeposit 的几个字段，包括 OrderID、Hash 和 index，都没有经过严格的检查。正因为如此，攻击者可以制作一连串的 MsgDeposit 消息，每个消息都有一个独有的 OrderID 和不同的 hash/index。该句柄将遍历并创建一连串的 NewOrderCollect 订单，然后将其保存到 order 保存器（keeper）中。

```
53      ...
54    if keeper.ok.IsExist(ctx, orderID) {
55      return sdk.ErrInvalidOrder(fmt.Sprintf("order %v already exists", orderID)).Result()
56    }

58    if keeper.ck.IsDepositExist(ctx, symbol.String(), toCUAddr, hash, index) {
59      return sdk.ErrInvalidTx(fmt.Sprintf("deposit %v %v %v %v item already exist", symbol
          , toCU, hash, index)).Result()
60    }

62    //ProcessOrder should be optimized.
63    processOrderList := keeper.ok.GetProcessOrderListByType(ctx, sdk.OrderType_Collect)
64    for _, id := range processOrderList {
65      order := keeper.ok.GetOrder(ctx, id)
66      if order != nil {
67        collectOrder := order.(*sdk.OrderCollect)
68        if collectOrder.Txhash == hash && collectOrder.Index == index {
69          return sdk.ErrInvalidTx(fmt.Sprintf("Tx: %v is already exist and not finish",
              hash)).Result()
70        }
71      }
72    }

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

Listing 3.18: bhchain/x/transfer/keeper/deposit.go

值得一提的是，transfer 模块也会处理 handleMsgConfirmedDeposit 消息。但是，所有的无效订单最终都没有被删除，只是其状态被更新为 Finish。因此，这会造成资源浪费或效率低

下。当累积的无效订单占据了所有的存储空间，最终会危害到各种正常的链上操作。

```
163 func (keeper BaseKeeper) confirmDepositOrder(ctx sdk.Context, order *sdk.OrderCollect,
        valid bool) ([]sdk.Flow, error) {
164   var flows []sdk.Flow
165   order.DepositStatus = sdk.Deposit_Confirmed
166   if !valid {
167     order.SetOrderStatus(sdk.OrderStatus_Finish)
168     return flows, nil
169   }
```

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

修复方法 在 handleMsgConfirmedDeposit 中规律性地删除无效订单。

3.15 在 SignedTx 验证中缺少错误处理（error handling）

- ID: PVE-015
- 危害性: 中
- 可能性: 低
- 影响力: 高
- 目标: `transfer`
- 类型: Coding Practices [16]
- CWE 子类: CWE-1071 [2]

描述

虽然 transfer 模块可以识别并处理 21 种不同的类型的消息，但这些消息通常围绕着用户存款、提款、内部收款、opcu 资产转移以及手续费转账（仅针对基于账户模型的代币类型）的业务逻辑。内部处理逻辑通常遵循四个不同的阶段：Begin、WaitSign、SignFinish 和 Finish。第一个阶段表示准备开始；第二个阶段调用密钥签署方法，并要求本轮验证者的响应；第三个阶段完成密钥签署过程，以便将签名后的交易进行广播和打包；第四个阶段意味着成功完成并达成共识，并完成必要的状态更新。

在本节中，我们将研究一个经常被调用的方法（verifyAccountBasedSignedTx），它经常被用于第三阶段，它可以校验当前验证者的签名。这个方法非常重要，因为它是阻止任何恶意破坏密钥签名行为的必要条件。然而，在该方法中（见下面的代码片段），它遗漏了一个错误处理（第 112 行）。我们认为，方法 chaininnode.QueryAccountTransactionFromData(chain, symbol, rawData) 的输入包含了不被信任的 rawdata，因此其返回值应该被全面校验。当返回值 rawTx 为 nil 的情况下，立即访问它会导致空指针错误（null pointer dereference），最终造成运行中断。

```
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)
99      if err != nil  !verified {
100         return sdk.ErrInvalidTx(fmt.Sprintf("VerifyAccountSignedTransaction fail:%v, err
                :%v", signedTx, err)).Result(), txHash
101     }

103     tx, err := keeper.cn.QueryAccountTransactionFromSignedData(chain, symbol, signedTx)
104     if err != nil {
105         return sdk.ErrInvalidTx(fmt.Sprintf("QueryUtxoTransactionFromSignedData Error:%v
                ", signedTx)).Result(), txHash
106     }

108     if tx.From != fromAddr {
109         return sdk.ErrInvalidTx(fmt.Sprintf("from an unexpected address:%v, expected
                address:%v", tx.From, fromAddr)).Result(), txHash
110     }

112     rawTx, _, err := keeper.cn.QueryAccountTransactionFromData(chain, symbol, rawData)
113     if tx.To != rawTx.To {
114         return sdk.ErrInvalidTx(fmt.Sprintf("to an unexpected address:%v, expected
                address:%v", tx.To, rawTx.To)).Result(), txHash
115     }

117     if !tx.Amount.Equal(rawTx.Amount) {
118         return sdk.ErrInvalidTx(fmt.Sprintf("amount mismatch,expected:%v, actual:%v",
                rawTx.Amount, tx.Amount)).Result(), txHash
119     }

121     if !tx.GasPrice.Equal(rawTx.GasPrice) {
122         return sdk.ErrInvalidTx(fmt.Sprintf("gasPrice mismatch, expected:%v, actual:%v",
                 rawTx.GasPrice, tx.GasPrice)).Result(), txHash
123     }

125     if !tx.GasLimit.Equal(rawTx.GasLimit) {
126         return sdk.ErrInvalidTx(fmt.Sprintf("gasLimit mismatch, expected:%v, actual:%v",
                 rawTx.GasLimit, tx.GasLimit)).Result(), txHash
127     }
128     txHash = tx.Hash

130     return sdk.Result{}, txHash
131 }
```

Listing 3.20: bhchain/x/transfer/keepers/utils.go

修复方法 在 verifyAccountBasedSignedTx 中增加必要的错误处理。

3.16 MsgSend/MsgMultiSend 中的黑洞收款地址

- ID: PVE-016
- 危害性: 参考
- 可能性: 低
- 影响力: 低
- 目标: `transfer`
- 类型: Coding Practices [16]
- CWE 子类: CWE-684 [12]

描述

除了前面几节讨论的消息类型外，transfer 模块还处理 MsgSend 和 MsgMultiSend 消息，旨在 HBTC Chain 中的地址之间转账。但是，这些消息的接收地址没有经过严格和彻底的有效性检查。因此，用户输入错误可能产生一个不合规（或不合法）的接收地址，纵使地址格式无效，句柄仍然会将资金发送到这些不合规的地址，最终导致资金无法收回。

```
52 // ValidateBasic Implements Msg.
53 func (msg MsgSend) ValidateBasic() sdk.Error {
54   if msg.FromAddress.Empty() {
55     return sdk.ErrInvalidAddress("missing sender address")
56   }
58   if msg.ToAddress.Empty() {
59     return sdk.ErrInvalidAddress("missing receipt address")
60   }
62   if !msg.Amount.IsValid() {
63     return sdk.ErrInvalidCoins("send amount is invalid: " + msg.Amount.String())
64   }
65   if !msg.Amount.IsAllPositive() {
66     return sdk.ErrInsufficientCoins("send amount must be positive")
67   }
68   return nil
69 }
```

Listing 3.21: bhchain/x/transfer/types/mesgs.go

修复方法 增加额外的检查以确保 MsgSend 和 MsgMultiSend 中的收款地址都符合 HBTC Chain 中标准的地址形式。

```
52 // ValidateBasic Implements Msg.
53 func (msg MsgSend) ValidateBasic() sdk.Error {
54   if msg.FromAddress.Empty()  !msg.FromAddress.IsValidAddr()  {
55     return sdk.ErrInvalidAddress("missing or wrong sender address")
56   }

58   if msg.ToAddress.Empty()  !msg.ToAddress.IsValidAddr() {
59     return sdk.ErrInvalidAddress("missing or wrong receipt address")
60   }

62   if !msg.Amount.IsValid() {
63     return sdk.ErrInvalidCoins("send amount is invalid: " + msg.Amount.String())
64   }
65   if !msg.Amount.IsAllPositive() {
66     return sdk.ErrInsufficientCoins("send amount must be positive")
67   }
68   return nil
69 }
```

Listing 3.22: bhchain/x/transfer/types/mesgs.go (revised)

3.17 在 Chainnode 中未能识别从合约账户发起的 ETH 存款

- ID: PVE-017
- 危害性: 中
- 可能性: 中
- 影响力: 中
- 目标: `chainnode`
- 类型: Business Logic [17]
- CWE 子类: CWE-841 [13]

描述

HBTC Chain 有一个 chainnode 组件，它主动监听外部区块链（例如 BTC 或者 Ethereum）并与之进行同步。该组件负责及时识别用户存款请求，并启动后续收款流程。

我们在下面展示了其作为 Ethereum 适配器（adaptor）的一部分代码片段。它不仅可以识别 ETH 的存款请求，还可以识别 ERC-20 代币的存款请求。但是，ETH 存款只识别了其外部交易的目的地址（toAddress.String()）。换句话说，其他的来自内部交易的存款将被忽略，且不会被接受。ERC-20 资产的存款则可以按照标准的 ERC-20 事件正确识别，不论其是内部交易还是由 EOA 发起的外部交易。

因此，当前的代码逻辑可能会错过通过内部交易的 ETH 存款。考虑到与各种 DeFi 协议合作的智能钱包越来越受欢迎，可能需要重新修改代码以适应这些智能钱包的 ETH 存款，因为这些存款请求通常是内部交易的一部分。

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

599         if tx.Value().Cmp(big.NewInt(0)) == 1 {
600           replyCh <- &proto.QueryAccountTransactionReply{
601             TxHash:          tx.Hash().String(),
602             TxStatus:        proto.TxStatus_Success,
603             From:            sender.String(),
604             To:              toAddress.String(),
605             Amount:          tx.Value().String(),
606             Memo:            "",
607             Nonce:           tx.Nonce(),
608             GasLimit:        new(big.Int).SetUint64(tx.Gas()).String(),
609             GasPrice:        tx.GasPrice().String(),
610             CostFee:         costFee.String(),
611             BlockHeight:     uint64(height),
612             BlockTime:       block.Time(),
613             SignHash:        signer.Hash(tx).Bytes(),
614             ContractAddress: "",
615           }
616         }
617         for _, receiptLog := range receipt.Logs {
618           if receiptLog.Removed {
619             continue
620           }
621           if len(receiptLog.Topics) != 3 {
622             continue
623           }
624           if receiptLog.Topics[0] != common.HexToHash("0
                xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef") {
625             continue
626           }

628           tokenFromAddress := common.BytesToAddress(receiptLog.Topics[1].Bytes())
629           tokenToAddress := common.BytesToAddress(receiptLog.Topics[2].Bytes())
630           tokenAmount, ok := big.NewInt(0).SetString(fmt.Sprintf("%x", receiptLog.Data),
                16)
631           if !ok {
632             errCh <- errors.New("failed to decode token amount from receipt log data")
633             needStop.Store(true)
634             return
635           }
636           if tokenAmount.Cmp(big.NewInt(0)) == 1 {
637             replyCh <- &proto.QueryAccountTransactionReply{
638               TxHash:          tx.Hash().String(),
639               TxStatus:        proto.TxStatus_Success,
640               From:            tokenFromAddress.String(),
641               To:              tokenToAddress.String(),
642               Amount:          tokenAmount.String(),
643               Memo:            "",
644               Nonce:           tx.Nonce(),
645               GasLimit:        new(big.Int).SetUint64(tx.Gas()).String(),
```

```
646             GasPrice:         tx.GasPrice().String(),
647             CostFee:          costFee.String(),
648             BlockHeight:      uint64(height),
649             BlockTime:        block.Time(),
650             SignHash:         signer.Hash(tx).Bytes(),
651             ContractAddress:  receiptLog.Address.String(),
652           }
653         }

655       }
```

Listing 3.23: chainnode/chainadaptor/ethereum/ethereum.go

同时，为了防止所谓的假充值（fake deposit），我们强烈建议对当前的区块高度和之前的区块高度（即区块高度-1）进行余额差值检查。任何不一致的情况都需要手动跟进并解决。

修复方法 为了更好地适配 DeFi，我们推荐接受由智能合约发起的 ETH 存款请求。

3.18 在密钥管理中移除不合作成员 (non-cooperating member)

- ID: PVE-018
- 危害性: 参考
- 可能性: 中
- 影响力: 未知
- 目标: `keymanager`
- 类型: Business Logic [17]
- CWE 子类: CWE-841 [13]

描述

在 HBTC Chain 中，settle 组件负责处理存款请求以及各种（分布式的）密钥生成和签名请求。它还以分布式的方式与当前活跃的验证者进行协作，并生成和签署共享密钥。如果某一个验证者不合作时，它可能会削弱当前多方密钥生成机制和签名协议的速度和健壮性。

该协议假定了一个威胁模型，即不诚信的大多数（dishonest majority），这意味着被恶意的用户或验证者的数量可以高达 n - 1，其中 n 是当前 epoch 中的活跃验证者数量。换句话说，尽管活跃度（liveness）对于密钥生成和签名协议非常重要，但此协议并不能保证它可以完成。如果没有活跃度，被保管的资产可能无法被使用，从而导致拒绝服务攻击。

多方密钥生成和签署与 HBTC Chain 的活跃度要求之间的错位，使得我们有必要及时发现并惩罚不合作的验证者。我们发现密钥生成和签署的各个阶段都可能暴露出不法行为者。因此，必须制定和运用激励机制，确保验证者在密钥生成和签署两个阶段都可以紧密且积极地配合，共同维护网络并履行自己的职责。同时，我们可以对非合作方的无意或有意的错误行为进行辨别。对于无意的行为，可能是由于构架问题或临时网络问题导致其无法参与密钥生成和签名；对于有意的行为，可以被认为是恶意行为，我们认为有必要采取主动且积极的惩罚措施。

因此，对有错误行为的验证者进行曝光是必要的，但这需要与内置的 slashing 模块直接联动。目前，仍然缺少所需的联动。

修复方法 在密钥管理中，对于合作者给予奖励，对于不合作者给予必要的惩罚措施。

3.19 生成正确的素数

- ID: PVE-019
- 危害性: 中
- 可能性: 中
- 影响力: 中
- 目标: `dsign`
- 类型: Coding Practices [16]
- CWE 子类: N/A

描述

在收到密钥生成请求后，settle 守护进程会启动去中心化的密钥生成协议，为参与方（即验证者）生成共享密钥。假设该协议在 n 方之间运行：P1，...，Pn，各方在输入阈值 t 和选定的椭圆曲线参数上运行。它通常有以下三轮：

- `Commitment Round:` 每一方 i 随机产生一个数字 ui，并向随机点 Yi = ui G 广播一个承诺（commit）；随后，每一方向 Yi 广播相应的解承诺（decommitment），这样每一方可以独立验证收到的 n-1 个解承诺的正确性。如果有任何不一致的地方，协议就会被中止。

- `VSS Round:` 每一方 i 都以值 ui 参与到 (t,n) Feldman VSS 方案中。整个群组的公钥是 Y（need latex here），i 的共享私钥是 xi（need latex here）。每一方 j 都随机地选取一个多项式的系数并秘密地将这个多项式结果发送给 i。而系数对于定义一个多项式是至关重要的，且是 sssa 的核心。为了正确地通知其他参与方他们最终的共享密钥，参与方 i 会广播一个零知识证明 xi（通过离散算法）。每一方都可以独立地验证其他的 n-1 个证明，如果证明失败，那么整个协议都会被中止。

- `Paillier KeyGen Round:` 每一方 i 在产生一个 Paillier 密钥对时，会广播公钥 ei。在这个机制的背后，每一方都会生成 Paillier 密钥对所需的安全素数 pi 和 qi，并广播他们对 pi , qi 的零知识证明，使 Ni = pi * qi（Ni 是与 Paillier 加密中的 RSA 算法的模数)。类似地，每一方都独立地验证 n-1 个接收到的证明，验证失败就中止。

我们需要留意平方根攻击（square root attack）可能造成的影响，这些攻击可能会影响 Paillier KeyGen Round。具体来说，计算任意组（质序的）离散对数的最佳算法是 baby-step giant-step 法、rho 法和 kangaroo 法。这些方法在时间和空间复杂度上有不同的取舍。为了避免这些攻击，最好的做法是确保 RSA 算法中的质数 pi ， qi 显著不同（通常是1024位）。

然而，RSAParameter() 中生成的质数似乎并没有遵循上述的标准。而它确实已经有一些检查来确保生成的 PTilde 和 QTilde 不完全相同（下面代码片段中的第 30 行）。然而，确保它们之间的差异足够大对于避免上述的平方根攻击也同样重要。

```
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   }

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)
41     }
42   }
43   p := big.NewInt(0).Rsh(big.NewInt(0).Sub(PTilde, one), 1)
44   q := big.NewInt(0).Rsh(big.NewInt(0).Sub(QTilde, one), 1)

46   NTilde := big.NewInt(0).Mul(PTilde, QTilde)
47   t1 := big.NewInt(0).ModInverse(QTilde, PTilde)
48   t2 := big.NewInt(0).ModInverse(PTilde, QTilde)
49   b01 := big.NewInt(0).Mul(big.NewInt(0).Mul(gP, t1), QTilde)
50   b02 := big.NewInt(0).Mul(big.NewInt(0).Mul(gQ, t2), PTilde)
51   b0 := big.NewInt(0).Mod(big.NewInt(0).Add(b01, b02), NTilde)

53   ...

55   return NTilde, PTilde, QTilde, h1, h2, nil
56 }
```

Listing 3.24: dsign/primes/primes.go

修复方法 我们强烈建议确保生成的素数具有所需的质量和长度。特别是，Paillier 加密中计算 N = pi*qi 时所用的两个素数 pi 和 qi 也需要确保差值（pi - qi）足够大（比如1020位），

以避免平方根攻击。

3.20 在 sssa.Create() 中不受限的私钥范围

- ID: PVE-020
- 危害性: 中
- 可能性: 低
- 影响力: 高
- 目标: sssa
- 类型: Arg.s and Parameters [19]
- CWE 子类: N/A

描述

HBTC Chain 在创建、部署和管理验证者之间的密钥共享方面做出了独特的贡献，以实现跨链资产及其之间的互相转换。密钥共享是基于已知的 Shamir 密钥共享算法（sssa）开发的。sssa 背后的思想是它需要 k+1 个点才能定义一个 k 阶的多项式，例如，2 个点定义一条直线，3 个点定义一条抛物线，4 个点定义一条三次方曲线并以此类推。

对于 (k, n) 阈值方案共享我们的密钥 S，sssa 算法通常会选择一个阶为 k-1 的随机多项式并伴有一个随机常数为密钥。该多项式不取零为最高项的系数。另外，多项式通常在大小为 P 的有限域 F 中运行，其中 0 < k <= n < P；S < P，且 P 为大质数。

```
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 3.25: dsign/dstsign/multisign.go

我们认为用于密钥共享的密钥 S 需要小于用于模运算的基数 P，即 S < P，如果我们按照密钥生成的程序执行的顺序，可能会动态生成一个私钥（在上面第 61 行的 keyGen 函数中），并直接传递给 sssa 用于共享密钥的生成。在 sssa 算法的内部，并没有采取任何检查来确保 S < P，缺少这一步检查有可能破坏共享密钥的生成，并可能导致秘钥丢失且不可恢复。

修复方法 在生成 Shamir 共享密钥时执行 S < P 的检查。

3.21 使用 0 填充密钥的临时值

- ID: PVE-021
- 危害性: 低
- 可能性: 低
- 影响力: 低
- 目标: `KeyManager`
- 类型: Coding Practices [16]
- CWE 子类: CWE-1091 [3]

描述

在密码学中，敏感参数（如加密私钥或口令）的使用通常会留下内存痕迹（memory footprint），这些痕迹应该被更好地清除。清零是常用的做法，即从加密模块中删除这些敏感参数（如加密私钥、口令和关键安全参数）以防止其泄露。

我们研究了一些方法，它们内部的计算大量基于这些敏感参数，并且需要应用上述的清零操作。一个例子是文件 key_gen.go 中的 keyGenHandler 中的 handleBeginMsg 方法。该方法是密钥生成服务的一部分，第 247 行中的局部变量 keyShare 包含了它本地的共享密钥。当它被使用后，建议将其赋值为零来清除使用痕迹。

```
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())

250     if err != nil {
251         return session, err
252     }

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

264     NTilde, PTilde, QTilde, h1, h2, err := primes.RSAParameter(RSALength)
265     if err != nil {
266         return session, err
267     }
268     trueShare := &dstsign.HonestShare{}
269     trueSchnorr := &dstsign.HonestSchnorr{}
270     truePQProof := &dstsign.HonestPQProof{}

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

275     _, tempNodeKey, _, _, _, err := dstsign.GetPublicKey(label, int(session.
            signThreshold),
276         len(session.keyNodes), PQProofK, bhcoreComm, maxRand, coeff, NTilde, PTilde,
                QTilde, h1, h2,
277         trueShare, trueSchnorr, truePQProof, keyShare)
278     session.nodeKey = tempNodeKey
279     if err == nil {
280         session.nodeKey.KeyNodes = session.keyNodes
281     }

283     return session, err
284 }
```

Listing 3.26: settle/keymanager/key_gen.go

类似的问题还可以在 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 中找到.

我们发现 settle 守护进程支持使用环境变量来传递敏感的密码信息（通过 passphrase := os.Getenv("PASSWORD")）。虽然使用起来很方便，但这种用法只应该在调试或测试环境中使

用，而不是在生产环境中使用。

修复方法 对于敏感的加密私钥和口令在使用之后立即清零。

3.22 在 MtAwc 中缺少有效性检查

- ID: PVE-022
- 危害性: 中
- 可能性: 中
- 影响力: 中
- 目标: `keymanager`
- 类型: Security Features [14]
- CWE 子类: CWE-285 [5]

描述

继上一节研究了 keygen 协议之后，本节我们分析 keysign 协议。如前所述，密钥签名协议严格遵循多方 ECDSA 算法。

这个算法有五个关键的相互依赖的阶段：commitment、MtA/MtAwc share conversion、delta(1) reconstruction、r generation 和 s generation。第一个阶段保证了会在接下来被频繁使用的随机数不可被否认。第二阶段利用加性同态加密机制，即 Paillier 加密，将乘性的共享密钥转换为加性的。这种转换是必要的，因为它确保了 t+1 方（而不是 2t+1 方）是足够生成最终的签名的。我们发现在第二个阶段的协议要求对两组随机数（从第一阶段中挑选出的）进行转换，每一对用户 Pi 和 Pj 都参与进了这个转换的子协议中：及 MtA 和 MtAwc。第三阶段重建 delta(-1)，这是四阶段计算 ECDSA 的随机数分量 r 所必需的。最后，第五阶段生成了 ECDSA 的签名分量：s。

如果我们仔细观察第二阶段，我们可以发现有两个转换过程：MtA 和 MtAwc。MtAwc 的不同之处在于为了确保参与方 Pi 是使用了正确的共享密钥值做了额外的检查，及 MtAwc with check。

```
847 func (t *Node) GetKeySignPhase2MsgSent(re Response) ([]SendingCheaterEvidence, error) {
848   t.KeySignPhase2MsgSent = make([]types.KeySignPhase2Msg, t.P-1)
849   errStr := ""
850   var evidenceList []SendingCheaterEvidence = make([]SendingCheaterEvidence, 0)
851   for k, v := range t.KeySignPhase1MsgReceived { //KeySignPhase1MsgReceived is not self-
          included
852     if !t.CheckSenderRangeProof(v.GetNativeSenderRangeProofK(), v.MessageK, v.
          GetNativePaillierPubKey()) {
853       errStr = errStr + v.LabelFrom + "K\n"
854       temp := SendingCheaterEvidence{v.LabelFrom, v.GetNativeSenderRangeProofK(), v.
            MessageK, v.GetNativePaillierPubKey()}
855       evidenceList = append(evidenceList, temp)
856     }
857     if !t.CheckSenderRangeProof(v.GetNativeSenderRangeProofR(), v.MessageR, v.
          GetNativePaillierPubKey()) {
858       errStr = errStr + v.LabelFrom + "R\n"
859       temp := SendingCheaterEvidence{v.LabelFrom, v.GetNativeSenderRangeProofR(), v.
            MessageR, v.GetNativePaillierPubKey()}
860       evidenceList = append(evidenceList, temp)
861     }
862     if errStr != "" {
863       continue
864     }
865     t.KeySignPhase2MsgSent[k].LabelFrom = t.label
866     t.KeySignPhase2MsgSent[k].LabelTo = v.LabelFrom
867     var Rk, Rr *big.Int
868     nTilde, h1, h2 := t.NTilde[v.LabelFrom], t.h1[v.LabelFrom], t.h2[v.LabelFrom]
869     pub := v.GetNativePaillierPubKey()
870     oneCipher, oneR := PaillierEnc(big.NewInt(1), pub)
871     t.KeySignPhase2MsgSent[k].MessageKResponse, Rk = getAnotherPart(v.MessageK, pub, t.
          randNumArray[k], t.r, oneCipher, oneR)
872     t.KeySignPhase2MsgSent[k].MessageRResponse, Rr = getAnotherPart(v.MessageR, pub, t.
          randNumArray[k], t.prtKey, oneCipher, oneR)
873     reR, rePrtKey := re.respond(t.r, t.prtKey)
874     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 3.27: settle/keymanger/multisign.go

上述代码片段展示了 GetKeySignPhase2MsgSent() 方法，这个方法准备了第二阶段中将要被使用的信息。MtA 和 MtAwc 的转换是在 getAnotherPart() 这个子方法中被执行的（在第 871 行和 872 行被调用了两次）。而 MtAwc 中所需的额外的检查并没有被执行。而缺少这个检查

很大程度上削弱了整个协议的安全性，例如参与者 Pi 可能提供一个不正确的密钥来误导最后签名的产生。

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

813   return cB, r
814 }
```

Listing 3.28: settle/keymanger/multisign.go

修复方法 确保 MtAwc 而不是 MtA执行了额外的检查。

4 结论

在本次安全审计中，我们分析了 HBTC Chain 和其相关的模块。 在审计的第一阶段，我们分析了项目源代码，并在代码库上运行我们的内部分析工具。 我们发现了一系列潜在的问题，而其中的一些涉及到多个模块之间的非正常交互。 而我们也相应地编写了各种测试用例来重现和验证每一个问题。 经过进一步的分析和内部讨论，我们确定有一些问题需要被提出并给予更多的关注，这些问题在第 2 章和第 3 章中进行了详细阐述。

通过这次审计，我们认为 HBTC Chain 具有良好的底层设计和工程能力。 代码库的组织逻辑清晰，模块实现优雅。我们识别出的问题都得到了及时的解决。 不论如何，HBTC Chain 的软件实现得非常好，并能迅速解决审计过程中发现的问题。 除此之外，正如第 1.4 节所述，我们欢迎任何关于本报告的建设性反馈或建议。

References

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

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

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

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

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

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

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

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

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

[10] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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