SOFTWARE AUDIT REPORT

for

HARMONY

Prepared By: Shuxiao Wang

Hangzhou, China
Jan. 08, 2020

Document Properties

Client	Harmony
Title	Software Audit Report
Target	Harmony Blockchain
Version	0.2
Author	Jeff Liu
Auditors	Edward Lo, Ruiyi Zhang, Huaguo Shi, Jeff Liu
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author	Description
0.2	Jan. 08, 2020	Jeff Liu	Add Two Findings
0.1	Sep. 30, 2019	Jeff Liu	Initial Draft

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 **Harmony Blockchain** design document and related source code, we in this report outline our systematic method to evaluate potential security issues in the Harmony Blockchain implementation, expose possible semantic inconsistency between the source code and the design specification, and provide additional suggestions and recommendations for improvement. Our results show that the given branch of Harmony Blockchain 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 Harmony Blockchain

Harmony [1] is a high performance, sharding-based blockchain developed by Harmony company, and its Day ONE mainnet was launched on June 28th, 2019. The goal of Harmony blockchain is to deliver scalability without sacrificing decentralization, with innovations in consensus, systems, and networking layers. Harmony uses a PBFT based consensus algorithm, named Fast Byzantine Fault Tolerance (FBFT), and PoS-based Sharding as a scalability solution. Harmony's randomness generation function is a combination of Verifiable Random Function (VRF) and Verifiable Delay Function (VDF).

The basic information of Harmony Blockchain is as follows:

Table 1.1: Basic Information of Harmony Blockchain

Item	Description
Issuer	Harmony
Website	https://harmony.one
Type	Harmony Blockchain
Platform	Go, C++, Solidity
Audit Method	White-box
Latest Audit Report	Jan. 08, 2020

The audited Git repositories and the commit hash values are as follows:

Table 1.2: The Commit Hash List Of Audited Branches

Git Repository	Commit Hash Of Audited Branch
https://github.com/harmony-one/harmony	de34b1753c825a24dc6448f2d513b29eec60d07d
https://github.com/harmony-one/vdf	b6aa89d16fd0d4f59b26c96dd1db6f35960222bf
https://github.com/harmony-one/bls	7d37e0af371482e08e32a7cb1f0a9d0a71d7b03f
https://github.com/harmony-one/ida	2993dd502a3de9d1aaa530717a334b8371539b32

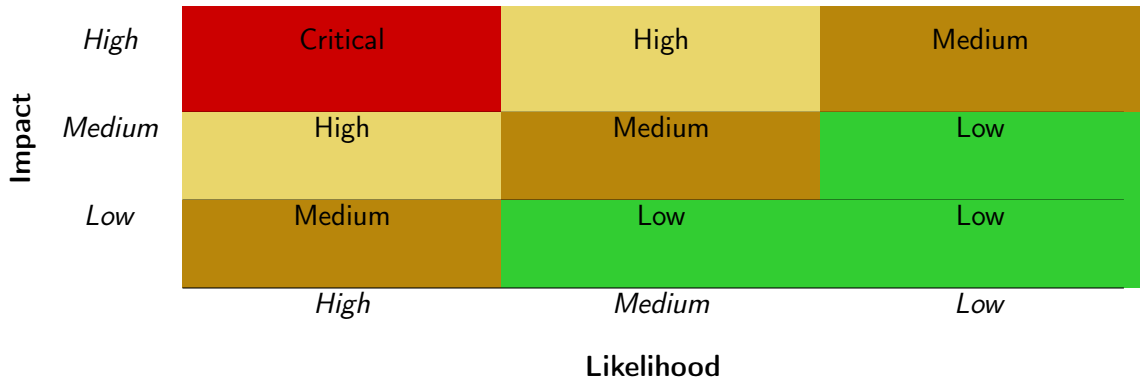
1.2 About PeckShield

PeckShield Inc. [2] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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 Harmony Blockchain, we use fuzzing to find out the corner cases NOT covered by in-house testing. Next we do white-box auditing, in which PeckShield security auditors manually review Harmony Blockchain design and source code, analyze them for any potential issues, also follow up with issues found in the fuzzing phase. We also design and implement test cases to further reproduce and verify the issues if necessary. In the following subsections, we will introduce the risk model as well as the audit procedure adopted in this report.

Table 1.3: Vulnerability Severity Classification



1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [3]:

- 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;

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*, *Low* shown in Table 1.3.

1.3.2 Fuzzing

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.

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by providing unintended input to the target program and monitoring the unexpected results. As one of the most effective methods for exploiting vulnerabilities, fuzzing technology has been the first choice for many security researchers to discover vulnerabilities in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to complete fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the Harmony Blockchain, we use AFL [4] and go-fuzz [5] 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 Harmony Blockchain, we will further analyze it as part of the white-box audit.

go-fuzz is a fuzzing tool inspired by AFL, for code written in Go language. It's a coverage guided fuzzing solution and mainly applicable to packages that parse complex inputs (both text and binary), and is especially useful for hardening of systems that parse inputs from potentially malicious users (e.g., anything accepted over a network).

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 we create specific test cases, execute them and analyze the results. Issues such as internal security holes, unexpected output, broken or poorly structured paths, etc., in the targeted software will be inspected.

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, during this audit we divide the blockchain software into the following major areas and inspect each of them:

- Data and state storage, which is related to the database and files where blockchain data are saved.
- P2P networking, consensus, and transaction model, which is the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.
- VM, account model, and incentive model. These are the execution and business layer of the blockchain, and many blockchain business specific logic is concentrated here.
- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.
- Others. Software modules not included above are checked here, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Table 1.4: The Full List of Audited Items

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
System Contracts And Services	System Contracts Security
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

Based on the above classification, here is the detailed list of the audited items as shown in Table 1.4.

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [6], 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 1.5 to classify our findings.

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.

Table 1.5: 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.

2 Findings

2.1 Finding Summary

Here is a summary of our findings after analyzing Harmony Blockchain. During the first phase of our audit, we studied Harmony source code and ran our in-house static code analyzer through the codebase, focused on the Harmony VM and crypto libraries. Next, we audited the general token transfer, staking, and consensus logics. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tools. We further manually review business logics, examine system operations, and place operation specific aspects under scrutiny to uncover possible pitfalls and/or bugs. 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 2 issues of that need to be brought up and pay more attention to, which are categorized in the table 2.1. More information can be found in the next subsection.

Here we also include screenshots of the current status of fuzzing. Figure 2.1 is a screenshot of a running AFL fuzzer which is testing the `bls` library. And, Figure 2.4 is the screenshot of a running Go-fuzz fuzzer which is testing the Harmony VM. We examine these parameters regularly, and whenever the *uniq crashes* increases, we look into the input which triggers the new unique crash. Once an issue that triggers crash is determined to be valid, further investigation will follow to root-cause and formulate fix recommendation for it.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Missing Sanity Check When Adding Cross Shard Receipts	Coding Practices	Fixed
PVE-002	Informational	Missing Penalty When Leaders Not Processing Cross Shard Receipts	Behavioral Issues	Confirmed

2.2 Key Findings

We conducted our audit of the Harmony design and implementations, starting with Harmony VM and crypto libraries, after that we audited general token transfer, staking, and consensus logics. After analyzing all of the potential issues found during the audit, we determined that a number of them need to be brought up and pay more attention to, as shown in Table 2.1. Please refer to Section 3 for detailed discussion of each vulnerability.

Harmony's VM is fully compatible with Ethereum VM (Constantinople), and they plan to support Wasm after mainnet launch. We worked through the Harmony VM code, and didn't find any fix missing for known Ethereum VM issues. We fed the Harmony VM through the go-fuzz tool, found two crashes and later determined to be caused by timeout. Further investigation found that they were timing issues related to go-fuzz, and there was no similar issue running Harmony VM directly. Therefore, we marked them as false warnings. The total coverage is pretty high, as shown in Figure 2.3, and the current status of the go-fuzz result is shown in Figure 2.4.

BLS signature scheme [7] is an excellent multisig solution which has some good properties compared to ECDSA [8] and Schnorr [9]. Harmony adopted the open source C++ BLS implementation [10] which has a harness that enables the integration with Golang software. We started our audit work with AFL fuzzing. Specifically, we used `afl-clang++` to compile the `bls` source code, which instruments the library as shown in Figure 2.2. Then, with a simple seed input, we started fuzzing the instrumented BLS as shown in Figure 2.1. During the first phase of our audit, we did not find any issue in the BLS library through AFL fuzzing. In the next phase, we will firstly try to improve the code coverage of fuzzing. Later, we will manually test and review the BLS implementation.

The other part of crypto libraries included in our first phase audit is the implementation of VDF, which is an essential component to provide trustworthy randomness on Harmony blockchain. With the trustworthy on-chain randomness, the blockchain would be able to safely support numerous applications such as dice dapps without an oracle mechanism. This is not a guaranteed feature on most blockchains. In many cases, the wrong implementations of on-chain mechanism caused tremendous financial damages [11, 12]. Our target here is a Golang implementation of Benjanmin Wesolowski's paper [13]. We started testing the library with the example `src/test/vdf_module_test.go` in this phase. In the next phase, we will apply go-fuzz on it as well.

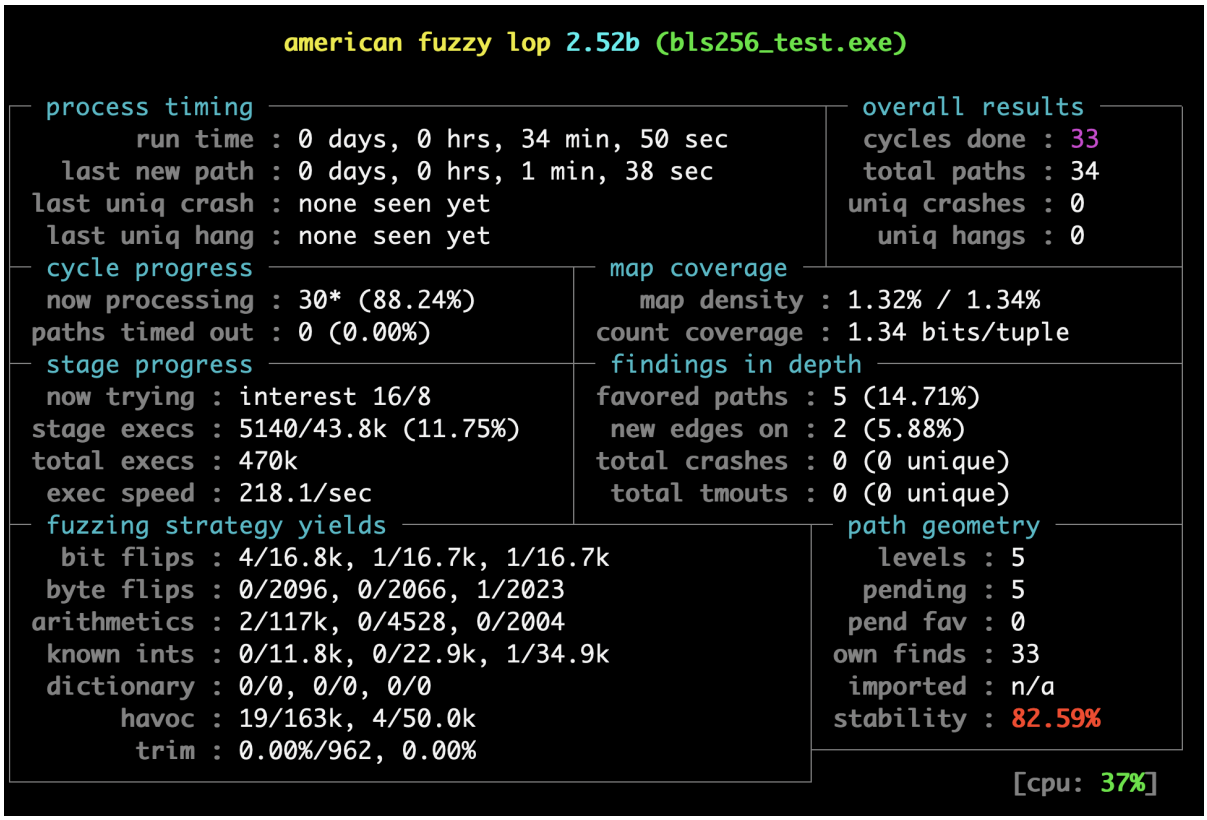


Figure 2.1: AFL Screenshot

```
[+] Instrumented 5630 locations (64-bit, non-hardened mode, ratio 100%).
ar r lib/libbls256.a obj/bls_c256.o
ar: creating archive lib/libbls256.a
../../afl-2.52b/afl-clang++ -shared -o lib/libbls256.dylib obj/bls_c256.o -L/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/lib -lmcl -lgmp -lgmpxx -L/usr/local/opt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcamtuf@google.com>
../../afl-2.52b/afl-clang++ -I/usr/local/opt/openssl/include -I/usr/local/opt/gmp/include -g3 -Wall -Wextra -Wformat=2 -Wcast-qual -Wcast-align -Wwrite-strings -Wfloat-equal -Wpointer-arith -m64 -I include -I test -fomit-frame-pointer -DNDEBUG -O3  -fPIC -std=c++11 -I/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/include -c src/bls_c384.cpp -o obj/bls_c384.o -MMD -MP -MF obj/bls_c384.d
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 5631 locations (64-bit, non-hardened mode, ratio 100%).
ar r lib/libbls384.a obj/bls_c384.o
ar: creating archive lib/libbls384.a
../../afl-2.52b/afl-clang++ -shared -o lib/libbls384.dylib obj/bls_c384.o -L/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/lib -lmcl -lgmp -lgmpxx -L/usr/local/opt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcamtuf@google.com>
../../afl-2.52b/afl-clang++ -I/usr/local/opt/openssl/include -I/usr/local/opt/gmp/include -g3 -Wall -Wextra -Wformat=2 -Wcast-qual -Wcast-align -Wwrite-strings -Wfloat-equal -Wpointer-arith -m64 -I include -I test -fomit-frame-pointer -DNDEBUG -O3  -fPIC -std=c++11 -I/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/include -c src/bls_c384_256.cpp -o obj/bls_c384_256.o -MMD -MP -MF obj/bls_c384_256.d
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 5631 locations (64-bit, non-hardened mode, ratio 100%).
ar r lib/libbls384_256.a obj/bls_c384_256.o
ar: creating archive lib/libbls384_256.a
../../afl-2.52b/afl-clang++ -shared -o lib/libbls384_256.dylib obj/bls_c384_256.o -L/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/lib -lmcl -lgmp -lgmpxx -L/usr/local/opt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcamtuf@google.com>
../../afl-2.52b/afl-clang++ -I/usr/local/opt/openssl/include -I/usr/local/opt/gmp/include -g3 -Wall -Wextra -Wformat=2 -Wcast-qual -Wcast-align -Wwrite-strings -Wfloat-equal -Wpointer-arith -m64 -I include -I test -fomit-frame-pointer -DNDEBUG -O3  -fPIC -std=c++11 -I/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/include -c src/bls_c512.cpp -o obj/bls_c512.o -MMD -MP -MF obj/bls_c512.d
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 5620 locations (64-bit, non-hardened mode, ratio 100%).
ar r lib/libbls512.a obj/bls_c512.o
ar: creating archive lib/libbls512.a
../../afl-2.52b/afl-clang++ -shared -o lib/libbls512.dylib obj/bls_c512.o -L/Users/cwu10/harmony-one/harmony-bls-work/bls/../mcl/lib -lmcl -lgmp -lgmpxx -L/usr/local/opt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcamtuf@google.com>
```

Figure 2.2: AFL Instrumentation

```
/go_project/src/github.com/harmony-one/harmony/core/vm/analysis.go (100.0%)
/go_project/src/github.com/harmony-one/harmony/core/vm/common.go (95.5%)
/go_project/src/github.com/harmony-one/harmony/core/vm/contract.go (97.7%)
/go_project/src/github.com/harmony-one/harmony/core/vm/contracts.go (83.7%)
/go_project/src/github.com/harmony-one/harmony/core/vm/evm.go (83.7%)
/go_project/src/github.com/harmony-one/harmony/core/vm/gas.go (87.5%)
/go_project/src/github.com/harmony-one/harmony/core/vm/gas_table.go (72.5%)
/go_project/src/github.com/harmony-one/harmony/core/vm/gen_structlog.go (6.2%)
/go_project/src/github.com/harmony-one/harmony/core/vm/instructions.go (98.4%)
/go_project/src/github.com/harmony-one/harmony/core/vm/interpreter.go (85.9%)
/go_project/src/github.com/harmony-one/harmony/core/vm/intpool.go (95.8%)
/go_project/src/github.com/harmony-one/harmony/core/vm/logger.go (1.4%)
/go_project/src/github.com/harmony-one/harmony/core/vm/memory.go (60.6%)
/go_project/src/github.com/harmony-one/harmony/core/vm/memory_table.go (100.0%)
/go_project/src/github.com/harmony-one/harmony/core/vm/opcodes.go (55.6%)
/go_project/src/github.com/harmony-one/harmony/core/vm/runtime/env.go (66.7%)
/go_project/src/github.com/harmony-one/harmony/core/vm/runtime/fuzz.go (100.0%)
/go_project/src/github.com/harmony-one/harmony/core/vm/runtime/runtime.go (61.9%)
/go_project/src/github.com/harmony-one/harmony/core/vm/stack.go (61.9%)
/go_project/src/github.com/harmony-one/harmony/core/vm/stack_table.go (62.5%)
```

Figure 2.3: Go-fuzz Coverage

```
2019/09/24 11:27:12 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9956, execs: 138102682 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:15 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9956, execs: 138103049 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:18 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9956, execs: 138103263 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:21 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9956, execs: 138103580 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:24 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138103862 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:27 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138104160 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:30 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138104423 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:33 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138104667 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:36 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138104908 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:39 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138105158 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:42 workers: 4, corpus: 404 (99h14m ago), crashers: 2, restarts: 1/9957, execs: 138105386 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:45 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138105638 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:48 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138105903 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:51 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138106293 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:54 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138106773 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:27:57 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138107182 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:28:00 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138107628 (228/sec), cover: 2115, uptime: 167h53m
2019/09/24 11:28:03 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138108139 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:06 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138108457 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:09 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138108998 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:12 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138109500 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:15 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138109992 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:18 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138110665 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:21 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138111283 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:24 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138111991 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:27 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138112599 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:30 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9956, execs: 138113254 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:33 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138114256 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:36 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138115166 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:39 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138116265 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:42 workers: 4, corpus: 404 (99h15m ago), crashers: 2, restarts: 1/9957, execs: 138117273 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:45 workers: 4, corpus: 404 (99h16m ago), crashers: 2, restarts: 1/9957, execs: 138118339 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:48 workers: 4, corpus: 404 (99h16m ago), crashers: 2, restarts: 1/9957, execs: 138119197 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:51 workers: 4, corpus: 404 (99h16m ago), crashers: 2, restarts: 1/9956, execs: 138119852 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:54 workers: 4, corpus: 404 (99h16m ago), crashers: 2, restarts: 1/9956, execs: 138120372 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:28:57 workers: 4, corpus: 404 (99h16m ago), crashers: 2, restarts: 1/9956, execs: 138120871 (228/sec), cover: 2115, uptime: 167h54m
2019/09/24 11:29:00 workers: 4, corpus: 404 (99h16m ago), crashers: 2, restarts: 1/9956, execs: 138121201 (228/sec), cover: 2115, uptime: 167h54m
```

Figure 2.4: Go-fuzz Screenshot

3 Detailed Results

3.1 Missing Sanity Check When Adding Cross Shard Receipts

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `node/node.go`
- Category: Coding Practices [14]
- CWE subcategory: CWE-20 [15]

Description

There is a vulnerability in the P2P module, which could be exploited by attackers to slow down the processing of cross shard transfers.

```
134 func (node *Node) ProcessReceiptMessage(msgPayload []byte) {
135     cxp := types.CXReceiptsProof{}
136     if err := rlp.DecodeBytes(msgPayload, &cxp); err != nil {
137         utils.Logger().Error().Err(err).Msg("[ProcessReceiptMessage] Unable to Decode
            message Payload")
138         return
139     }
140     utils.Logger().Debug().Interface("cxp", cxp).Msg("[ProcessReceiptMessage] Add
        CXReceiptsProof to pending Receipts")
141     // TODO: integrate with txpool
142     node.AddPendingReceipts(&cxp)
143 }
```

Listing 3.1: node/node_cross_shard.go

ProcessReceiptMessage will be called for receipts messages. It will decode the cross shard receipts and merkle proof encoded in RLP format, and pass them to AddPendingReceipts (line 142).

```
332 func (node *Node) AddPendingReceipts(receipts *types.CXReceiptsProof) {
333   node.pendingCXMutex.Lock()
334   defer node.pendingCXMutex.Unlock()
335
336   if receipts.ContainsEmptyField() {
337     utils.Logger().Info().Int(... ...)
```

```
338       return
339     }
340
341     blockNum := receipts.Header.Number().Uint64()
342     shardID := receipts.Header.ShardID()
343     key := utils.GetPendingCXKey(shardID, blockNum)
344
345     if _, ok := node.pendingCXReceipts[key]; ok {
346       utils.Logger().Info().Int(... ...)
347       return
348     }
349     node.pendingCXReceipts[key] = receipts
350     utils.Logger().Info().Int(... ...)
351   }
```

Listing 3.2: node/node.go

```
183 // ContainsEmptyField checks whether the given CXReceiptsProof contains empty field
184 func (cxp *CXReceiptsProof) ContainsEmptyField() bool {
185     return cxp == nil || cxp.Receipts == nil || cxp.MerkleProof == nil || cxp.Header ==
        nil || len(cxp.CommitSig)+len(cxp.CommitBitmap) == 0
186 }
```

Listing 3.3: core/types/cx_receipt.go

AddPendingReceipts will first check whether the receipt contains empty fields (line 336) or had been recorded in the pendingCXReceipts map (line 345), and will save it if not (line 349).

However, there is no further sanity check enforced while adding new receipts into pendingCXReceipts. Specifically, a malicious attacker can craft a valid yet meaningless CXReceiptsProof and send it to the victims to occupy the pendingCXReceipts map with the key composed from shardID and blockNum, which will block the real CXReceiptsProof from normal nodes and slow down the cross shard transfer processing.

Recommendation Add sanity checks for the origin and validity of the cross shard receipts.

3.2 Missing Penalty When Leaders Not Processing Cross Shard Receipts

- ID: PVE-002
- Severity: Informational
- Likelihood: High
- Impact: None/Undetermined
- Target: `node/worker/worker.go`
- Category: Behavioral Issues [16]
- CWE subcategory: CWE-841 [17]

Description

The cross shard transfer is supported on harmony blockchain. The process can be summarized as follows:

1) Source shards run the cross shard transactions, and broadcast cross shard receipts to destination shards.

2) Destination shards receive the receipts and put them in a pending map.

3) Destination shards leaders handle the cross shard receipts in the new blocks.

```
79 func (node *Node) proposeNewBlock() (*types.Block, error) {
80   node.Worker.UpdateCurrent()
81 ... ...
```

Listing 3.4: node/node_newblock.go

```
124   if err := node.Worker.CommitTransactions(
125     pending, pendingStakingTransactions, beneficiary,
126     func(payload staking.RPCTransactionError) {
127       const maxSize = 1024
128       node.errorSink.Lock()
129       if l := len(node.errorSink.failedTxns); l >= maxSize {
130         node.errorSink.failedTxns = append(node.errorSink.failedTxns[1:], payload)
131       } else {
132         node.errorSink.failedTxns = append(node.errorSink.failedTxns, payload)
133       }
134       node.errorSink.Unlock()
135     },
136   ); err != nil {
137     utils.Logger().Error().Err(err).Msg("cannot commit transactions")
138     return nil, err
139   }
140
141   // Prepare cross shard transaction receipts
142   receiptsList := node.proposeReceiptsProof()
143   if len(receiptsList) != 0 {
144     if err := node.Worker.CommitReceipts(receiptsList); err != nil {
145       utils.Logger().Error().Err(err).Msg("[proposeNewBlock] cannot commit receipts")
146     }
147   }
```

Listing 3.5: node/node_newblock.go

`proposeNewBlock` is called by shard leaders for proposing a new block. It will process the pending transactions / staking transactions (line 124 - 139), and handle the cross shard transaction receipts (line 142 - 147).

```
206 func (w *Worker) CommitReceipts(receiptsList []*types.CXReceiptsProof) error {
207   if w.current.gasPool == nil {
208     w.current.gasPool = new(core.GasPool).AddGas(w.current.header.GasLimit())
209   }
210
211   if len(receiptsList) == 0 {
212     w.current.header.SetIncomingReceiptHash(types.EmptyRootHash)
213   } else {
214     w.current.header.SetIncomingReceiptHash(types.DeriveSha(types.CXReceiptsProofs(
          receiptsList)))
215   }
216
217   for _, cx := range receiptsList {
218     err := core.ApplyIncomingReceipt(w.config, w.current.state, w.current.header, cx)
219     if err != nil {
220       return ctxerror.New("cannot apply receiptsList").WithCause(err)
221     }
222   }
223
224   for _, cx := range receiptsList {
225     w.current.incxs = append(w.current.incxs, cx)
226   }
227   return nil
228 }
```

Listing 3.6: node/worker/worker.go

`CommitReceipts` will apply the receipts and adjust the balance of the corresponding account (line 218). However, there is no penalty if shard leader intentionally ignore any specific receipts and let them stay pending forever. Specifically, a leader is free to choose any receipts in the `node.pendingCXReceipts` map, not by timestamp or any other specific rule, and there is no penalty if a malicious leader intentionally ignore some receipts. Ttechnically, a leader can skip some cross shard receipts on purpose and let them stay pending forever.

Recommendation Add penalty when leaders do not process cross shard receipts. According to Harmony, leader rotation and the mechanisms to detect transaction withholding and preempt a malicious leader will be added in the next phase of mainnet upgrade. In the current phase where Harmony controls the leader nodes, this is not an issue to the users.

4 | Conclusion

For this security audit, we have analyzed the Harmony Blockchain. During the first phase of our audit, we studied the source code and ran our in-house analyzing tools through the codebase, including areas such as Harmony VM and crypto libraries. Next, we audited the general token transfer, staking, and consensus logics. A list of potential issues were found, and some of them involve unusual interactions among multiple modules, therefore we developed 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 pay more attention to, which are reported in Sections 2 and 3. Given that the reported issues have been fixed, we do feel that the Harmony VM and token transfer logic have been thoroughly inspected, and there is no other known issues in those areas, therefore they can be deployed on the blockchain with confidence.

Our impression through this audit is that the Harmony Blockchain software is neatly organized and elegantly implemented and those identified issues are promptly confirmed and fixed. We'd like to commend Harmony 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] Harmony. Harmony Inc. https://harmony.one.

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

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

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

[5] gofuzz. gofuzz. https://github.com/dvyukov/go-fuzz.

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

[7] Wikipedia. Boneh–Lynn–Shacham. https://en.wikipedia.org/wiki/Boneh%E2%80%93Lynn%E2%80%93Shacham.

[8] Wikipedia. Elliptic Curve Digital Signature Algorithm. https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[9] Wikipedia. Schnorr signature. https://en.wikipedia.org/wiki/Schnorr_signature.

[10] MITSUNARI Shigeo. An implementation of BLS threshold signature. https://github.com/herumi/bls.

[11] PeckShield. Pwning Fomo3D Revealed: Iterative, Pre-Calculated Contract Creation For Airdrop Prizes! https://blog.peckshield.com/2018/07/24/fomo3d/.

[12] PeckShield. Defeating EOS Gambling Games: The Techniques Behind Random Number Loophole. https://blog.peckshield.com/2018/11/22/eos/.

[13] Benjamin Wesolowski. Efficient verifiable delay functions. Advances in Cryptology – EUROCRYPT 2019, 11478:379–407, 2019.

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

[15] MITRE. CWE-20: Improper Input Validation. https://cwe.mitre.org/data/definitions/20.html.

[16] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

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