



SOFTWARE AUDIT REPORT

for

HARMONY



Prepared By: Shuxiao Wang

Hangzhou, China

Jul. 06, 2020

Document Properties

Client	Harmony
Title	Software Audit Report
Target	Harmony Blockchain
Version	0.4
Author	Jeff Liu
Auditors	Edward Lo, Ruiyi Zhang, Xudong Shao, Jeff Liu
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
0.4	Jul. 06, 2020	Jeff Liu	Update Finding Status
0.3	Apr. 22, 2020	Jeff Liu	More Findings Added
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	5
1.1	About Harmony Blockchain	5
1.2	About PeckShield	6
1.3	Methodology	6
1.3.1	Risk Model	7
1.3.2	Fuzzing	7
1.3.3	White-box Audit	8
1.4	Disclaimer	10
2	Findings	12
2.1	Finding Summary	12
2.2	Key Findings	14
3	Detailed Results	17
3.1	Missing Sanity Check When Adding Cross Shard Receipts	17
3.2	Missing Penalty When Leaders Not Processing Cross Shard Receipts	18
3.3	Out-of-Bounds Access in the P2P Module - #1	20
3.4	Out-of-Bounds Access in the P2P Module - #2	22
3.5	Out-of-Bounds Access in the P2P Module - #3	24
3.6	DoS Vulnerability in the P2P Module - #1	27
3.7	DoS Vulnerability in the P2P Module - #2	30
3.8	Integer Overflow in the RPC Module	33
3.9	Consensus Suspending in the Consensus Module - #1	35
3.10	Out-of-Memory in the Consensus Module - #1	37
3.11	Out-of-Memory in the Consensus Module - #2	40
3.12	Consensus Suspending in the Consensus Module - #2	44
3.13	Consensus Suspending in the Consensus Module - #3	46
3.14	Missing Sanity Check on Slash Records - #1	53
3.15	Missing Sanity Check on Slash Records - #2	57

4 Conclusion	60
References	61



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	Jul. 06, 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	00b3abe94f9b3c29c34723973a66943b1e9266a1
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

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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 Problems	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.
Input Validation Issues	Weaknesses in this category are related to a software system's input validation components.

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, as summarized in table 2.1, we determined 15 issues of that need to be brought up and pay more attention to, which are categorized in the table 2.2. 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 `b1s` 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





Severity	# of Findings	
Critical	10	
High	2	
Medium	2	
Low	0	
Informational	1	
Total	15	

Table 2.1: The Severity of Our Findings

issue that triggers crash is determined to be valid, further investigation will follow to root-cause and formulate fix recommendation for it.

Table 2.2: 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 Problems	Confirmed
PVE-003	Critical	Out-of-Bounds Access in the P2P Module - #1	Coding Practices	Fixed
PVE-004	Critical	Out-of-Bounds Access in the P2P Module - #2	Coding Practices	Fixed
PVE-005	Critical	Out-of-Bounds Access in the P2P Module - #3	Coding Practices	Fixed
PVE-006	Critical	DoS Vulnerability in the P2P Module - #1	Behavioral Problems	Fixed
PVE-007	Critical	DoS Vulnerability in the P2P Module - #2	Coding Practices	Fixed
PVE-008	Medium	Integer Overflow in the RPC module	Coding Practices	Fixed
PVE-009	Critical	Consensus Suspending in the Consensus Module - #1	Behavioral Problems	Fixed
PVE-010	Critical	Out-of-Memory in the Consensus Module - #1	Behavioral Problems	Fixed
PVE-011	Critical	Out-of-Memory in the Consensus Module - #2	Behavioral Problems	Fixed
PVE-012	Critical	Consensus Suspending in the Consensus Module - #2	Behavioral Problems	Fixed
PVE-013	High	Consensus suspending in the Consensus Module - #3	Input Validation Issues	Fixed
PVE-014	Critical	Missing Sanity Check on Slash Records - #1	Input Validation Issues	Fixed
PVE-015	High	Missing Sanity Check on Slash Records - #2	Behavioral Problems	Fixed

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 2.2. 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 Benjamin 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.

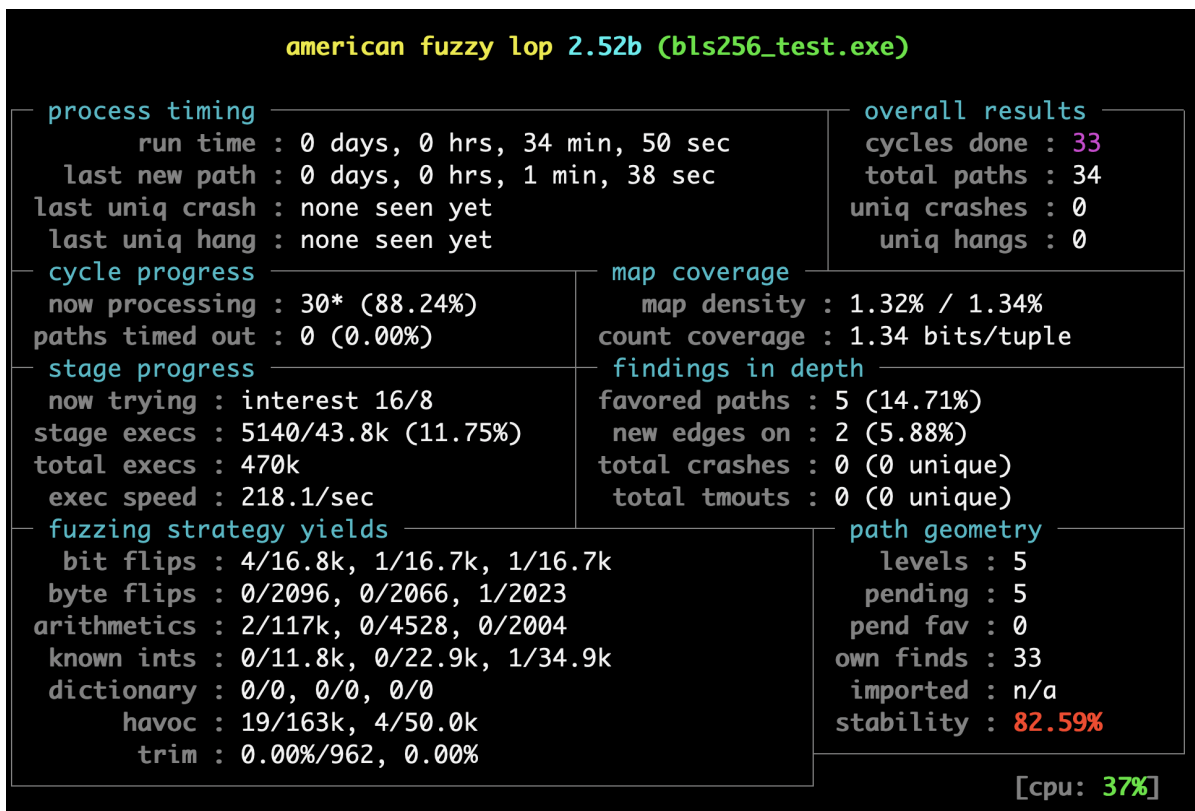


Figure 2.1: AFL Screenshot

```
[*] 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/o
pt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcantuf@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 -DDEBUG -O3 -fPIC -std=c++11 -I/Users/cwu10/harmony-one/harmony-bls-work/bls/./mcl/include -c sr
c/bls_c384.cpp -o obj/bls_c384.o -MMD -MP -MF obj/bls_c384.d
afl-cc 2.52b by <lcantuf@google.com>
afl-as 2.52b by <lcantuf@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/o
pt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcantuf@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 -DDEBUG -O3 -fPIC -std=c++11 -I/Users/cwu10/harmony-one/harmony-bls-work/bls/./mcl/include -c sr
c/bls_c384_256.cpp -o obj/bls_c384_256.o -MMD -MP -MF obj/bls_c384_256.d
afl-cc 2.52b by <lcantuf@google.com>
afl-as 2.52b by <lcantuf@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/lo
cal/opt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcantuf@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 -DDEBUG -O3 -fPIC -std=c++11 -I/Users/cwu10/harmony-one/harmony-bls-work/bls/./mcl/include -c sr
c/bls_c512.cpp -o obj/bls_c512.o -MMD -MP -MF obj/bls_c512.d
afl-cc 2.52b by <lcantuf@google.com>
afl-as 2.52b by <lcantuf@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/o
pt/openssl/lib -lcrypto -lstdc++
afl-cc 2.52b by <lcantuf@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     }
139     return
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 ==
186         nil || len(cxp.CommitSig)+len(cxp.CommitBitmap) == 0

```

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 Problems [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(
215             receiptsList)))
216     }
217
218     for _, cx := range receiptsList {
219         err := core.ApplyIncomingReceipt(w.config, w.current.state, w.current.header, cx)
220         if err != nil {
221             return ctxerror.New("cannot apply receiptsList").WithCause(err)
222         }
223     }
224
225     for _, cx := range receiptsList {
226         w.current.incxs = append(w.current.incxs, cx)
227     }
228     return nil
229 }

```

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. Technically, 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.

3.3 Out-of-Bounds Access in the P2P Module - #1

- ID: PVE-003
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: node/node_handler.go
- Category: Coding Practices [14]
- CWE subcategory: CWE-129 [18]

Description

This is a vulnerability in the P2P module, which could be exploited by attackers to perform DoS attack against the harmony network.

Within the harmony network, a node can be one of the these roles: validator, leader, beacon validator, or beacon leader depending on its context. With each role, a node would run a certain set of services.

Furthermore, harmony network has enabled libp2p based gossiping using pubsub. Nodes no longer send messages to individual nodes, instead, they publish / subscribe to different topics.

```

39 // receiveGroupMessage use libp2p pubsub mechanism to receive broadcast messages
40 func (node *Node) receiveGroupMessage(
41     receiver p2p.GroupReceiver, rxQueue msgq.MessageAdder,
42 ) {
43     ctx := context.Background()
44     // TODO ek - infinite loop; add shutdown/cleanup logic
45     for {
46         msg, sender, err := receiver.Receive(ctx)
47         if err != nil {
48             utils.Logger().Warn().Err(err).
49                 Msg("cannot receive from group")
50             continue
51         }
52         if sender == node.host.GetID() {
53             continue
54         }
55         //utils.Logger().Info("[PUBSUB]", "received group msg", len(msg), "sender",
56             sender)
57         // skip the first 5 bytes, 1 byte is p2p type, 4 bytes are message size
58         if err := rxQueue.AddMessage(msg[5:], sender); err != nil {
59             utils.Logger().Warn().Err(err).
60                 Str("sender", sender.Pretty()).
61                 Msg("cannot enqueue incoming message for processing")
62         }
63     }

```

Listing 3.7: node/node_handler.go

Specifically, each node will call `receiveGroupMessage` to receive broadcast messages, and distribute them to consumers. However, there is no sanity check for the length of the received messages. It simply passes the buffer start from offset 5 (line 57) to queues, which could cause out-of-bound access panic for nodes subscribe to the topic if the message length < 5 .

Recommendation Add sanity checks for the length of the received messages.

3.4 Out-of-Bounds Access in the P2P Module - #2

- ID: PVE-004
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: node/node_handler.go
- Category: Coding Practices [14]
- CWE subcategory: CWE-129 [18]

Description

This is a vulnerability in the P2P module, which could be exploited by attackers to perform DoS attack against the harmony network.

Within the harmony network, a node can be treated as one of the roles: validator, leader, beacon validator, or beacon leader depending on its context. With each role, a node can run a certain set of services.

Also, harmony has enabled libp2p based gossiping using pubsub. Nodes no longer send messages to individual nodes, instead, they publish / subscribe to different topics.

```

39 // receiveGroupMessage use libp2p pubsub mechanism to receive broadcast messages
40 func (node *Node) receiveGroupMessage(
41     receiver p2p.GroupReceiver, rxQueue msgq.MessageAdder,
42 ) {
43     ctx := context.Background()
44     // TODO ek - infinite loop; add shutdown/cleanup logic
45     for {
46         msg, sender, err := receiver.Receive(ctx)
47         if err != nil {
48             utils.Logger().Warn().Err(err).
49                 Msg("cannot receive from group")
50             continue
51         }
52         if sender == node.host.GetID() {
53             continue
54         }
55         //utils.Logger().Info("[PUBSUB]", "received group msg", len(msg), "sender",
56             sender)
57         // skip the first 5 bytes, 1 byte is p2p type, 4 bytes are message size
58         if err := rxQueue.AddMessage(msg[5:], sender); err != nil {
59             utils.Logger().Warn().Err(err).
60                 Str("sender", sender.Pretty()).
61                 Msg("cannot enqueue incoming message for processing")
62         }
63     }

```

Listing 3.8: node/node_handler.go

Each node would call `receiveGroupMessage` to receive broadcast messages, and distribute them to consumers. Messages are encoded as the following format:

```

1  ----  content start  ----
2  1 byte          - message category
3                      0x00: Consensus
4                      0x01: Node...
5  1 byte          - message type
6                      - for Consensus category
7                      0x00: consensus
8                      0x01: sharding ...
9                      - for Node category
10                     0x00: transaction ...
11 n - 2 bytes      - actual message payload
12 ----  content end  ----

```

Every message has its category and type, and would be handled accordingly.

```

66 func (node *Node) HandleMessage(content []byte, sender libp2p_peer.ID) {
67     msgCategory, err := proto.GetMessageCategory(content)
68     if err != nil {
69         utils.Logger().Error().
70             Err(err).
71             Msg("HandleMessage get message category failed")
72     }
73     return
74
75     msgType, err := proto.GetMessageType(content)
76     if err != nil {
77         utils.Logger().Error().
78             Err(err).
79             Msg("HandleMessage get message type failed")
80     }
81     return
82
83     msgPayload, err := proto.GetMessagePayload(content)
84     if err != nil {
85         utils.Logger().Error().
86             Err(err).
87             Msg("HandleMessage get message payload failed")
88     }
89     return
90
91     switch msgCategory {
92     case proto.Consensus:
93         msgPayload, _ := proto.GetConsensusMessagePayload(content)
94         if node.NodeConfig.Role() == nodeconfig.ExplorerNode {
95             node.ExplorerMessageHandler(msgPayload)
96         } else {
97             node.ConsensusMessageHandler(msgPayload)
98         }
99     case proto.DRand:
100         msgPayload, _ := proto.GetDRandMessagePayload(content)
101         if node.DRand != nil {
102             if node.DRand.IsLeader {
103                 node.DRand.ProcessMessageLeader(msgPayload)

```

```

104         } else {
105             node.DRand.ProcessMessageValidator(msgPayload)
106         }
107     }
108     case proto.Node:
109         actionType := proto_node.MessageType(msgType)
110         switch actionType {
111             case proto_node.Transaction:
112                 utils.Logger().Debug().Msg("NET: received message: Node/Transaction")
113                 node.transactionMessageHandler(msgPayload)
114             case proto_node.Staking:
115                 utils.Logger().Debug().Msg("NET: received message: Node/Staking")
116                 node.stakingMessageHandler(msgPayload)
117             case proto_node.Block:
118                 utils.Logger().Debug().Msg("NET: received message: Node/Block")
119                 blockMsgType := proto_node.BlockMessageType(msgPayload[0])

```

Listing 3.9: node/node_handler.go

msgCategory, msgType, msgPayload are extracted from the message (msg[0], msg[1], msg[2:]), and HandleMessage will take different actions according to them. However, there is no sanity check for msgPayload for proto_node.Block case (line 117).

Specifically, if a malicious attacker passed in a small buffer (length = 7) with msgCategory = proto.Node and msgType = proto_node.Block, msgPayload (line 83) will be a 0 length slice, and accessing to it (line 119) would cause out-of-bound access panic for nodes subscribe to the topic.

Recommendation Add sanity checks for the length of the received messages.

3.5 Out-of-Bounds Access in the P2P Module - #3

- ID: PVE-005
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: node/node_handler.go
- Category: Coding Practices [14]
- CWE subcategory: CWE-129 [18]

Description

This is a vulnerability in the P2P module, which could be exploited by attackers to perform DoS attack against the harmony network.

Within the harmony network, a node can be treated as one of the roles: validator, leader, beacon validator, or beacon leader depending on its context. With each role, a node can run a certain set of services.

The harmony network has also enabled libp2p based gossiping using pubsub. Nodes no longer send messages to individual nodes, instead, they publish / subscribe to different topics.

```

39 // receiveGroupMessage use libp2p pubsub mechanism to receive broadcast messages
40 func (node *Node) receiveGroupMessage(
41     receiver p2p.GroupReceiver, rxQueue msgq.MessageAdder,
42 ) {
43     ctx := context.Background()
44     // TODO ek - infinite loop; add shutdown/cleanup logic
45     for {
46         msg, sender, err := receiver.Receive(ctx)
47         if err != nil {
48             utils.Logger().Warn().Err(err).
49                 Msg("cannot receive from group")
50             continue
51         }
52         if sender == node.host.GetID() {
53             continue
54         }
55         //utils.Logger().Info("[PUBSUB]", "received group msg", len(msg), "sender",
56             sender)
57         // skip the first 5 bytes, 1 byte is p2p type, 4 bytes are message size
58         if err := rxQueue.AddMessage(msg[5:], sender); err != nil {
59             utils.Logger().Warn().Err(err).
60                 Str("sender", sender.Pretty()).
61                 Msg("cannot enqueue incoming message for processing")
62         }
63     }
64 }

```

Listing 3.10: node/node_handler.go

Each node would call `receiveGroupMessage` to receive broadcast messages, and distribute them to consumers. Messages are encoded as the following format:

```

1  ---- content start ----
2  1 byte                - message category
3                        0x00: Consensus
4                        0x01: Node...
5  1 byte                - message type
6                        - for Consensus category
7                        0x00: consensus
8                        0x01: sharding ...
9                        - for Node category
10                       0x00: transaction ...
11 n - 2 bytes          - actual message payload
12 ---- content end ----

```

Every message has its category and type, and would be handled accordingly.

```

66 func (node *Node) HandleMessage(content []byte, sender libp2p_peer.ID) {
67     msgCategory, err := proto.GetMessageCategory(content)
68     if err != nil {

```

```

69     utils.Logger().Error().
70         Err(err).
71         Msg("HandleMessage get message category failed")
72     return
73 }
74
75 msgType, err := proto.GetMessageType(content)
76 if err != nil {
77     utils.Logger().Error().
78         Err(err).
79         Msg("HandleMessage get message type failed")
80     return
81 }
82
83 msgPayload, err := proto.GetMessagePayload(content)
84 if err != nil {
85     utils.Logger().Error().
86         Err(err).
87         Msg("HandleMessage get message payload failed")
88     return
89 }
90
91 switch msgCategory {
92 case proto.Consensus:
93     msgPayload, _ := proto.GetConsensusMessagePayload(content)
94     if node.NodeConfig.Role() == nodeconfig.ExplorerNode {
95         node.ExplorerMessageHandler(msgPayload)
96     } else {
97         node.ConsensusMessageHandler(msgPayload)
98     }
99 case proto.DRand:
100     msgPayload, _ := proto.GetDRandMessagePayload(content)
101     if node.DRand != nil {
102         if node.DRand.IsLeader {
103             node.DRand.ProcessMessageLeader(msgPayload)
104         } else {
105             node.DRand.ProcessMessageValidator(msgPayload)
106         }
107     }
108 case proto.Node:
109     actionType := proto_node.MessageType(msgType)
110     switch actionType {
111     case proto_node.Transaction:
112         utils.Logger().Debug().Msg("NET: received message: Node/Transaction")
113         node.transactionMessageHandler(msgPayload)
114     case proto_node.Staking:
115         utils.Logger().Debug().Msg("NET: received message: Node/Staking")
116         node.stakingMessageHandler(msgPayload)
117     case proto_node.Block:
118         utils.Logger().Debug().Msg("NET: received message: Node/Block")
119         blockMsgType := proto_node.BlockMessageType(msgPayload[0])

```

Listing 3.11: node/node_handler.go

msgCategory, msgType, msgPayload are extracted from the message (msg[0], msg[1], msg[2:]), and HandleMessage would take different actions according to them.

```

173 func (node *Node) transactionMessageHandler(msgPayload []byte) {
174     txMessageType := proto_node.TransactionMessageType(msgPayload[0])
175
176     switch txMessageType {
177     case proto_node.Send:
178         txs := types.Transactions{}
179         err := rlp.Decode(bytes.NewReader(msgPayload[1:]), &txs) // skip the Send message
180             type
181         if err != nil {
182             utils.Logger().Error().
183                 Err(err).
184                 Msg("Failed to deserialize transaction list")
185             return
186         }
187         node.addPendingTransactions(txs)
188     }
189 }

```

Listing 3.12: node/node_handler.go

transactionMessageHandler would be called for transaction messages. However, there is no sanity check for msgPayload (line 174).

To be exact, if a malicious attacker passed in a small buffer (length = 7) with msgCategory = proto.Node and msgType = proto_node.Transaction, msgPayload will be a 0 length slice, and accessing to it (line 174) could cause OOB access panic for nodes subscribe to the topic.

Recommendation Add sanity checks for the length of the received messages.

3.6 DoS Vulnerability in the P2P Module - #1

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: node/node.go
- Category: Behavioral Problems [16]
- CWE subcategory: CWE-696 [19]

Description

This is a vulnerability in the P2P module, which could be exploited by attackers to perform DoS attack against the harmony network.

Within the harmony network, a node can be one of the these roles: validator, leader, beacon validator, or beacon leader depending on its context. With each role, a node would run a certain set of services.

Furthermore, harmony network has enabled libp2p based gossiping using pubsub. Nodes no longer send messages to individual nodes, instead, they publish / subscribe to different topics.

```

66 func (node *Node) HandleMessage(content []byte, sender libp2p_peer.ID) {
67     msgCategory, err := proto.GetMessageCategory(content)
68     if err != nil {
69         utils.Logger().Error().
70             Err(err).
71             Msg("HandleMessage get message category failed")
72     }
73     return
74
75     msgType, err := proto.GetMessageType(content)
76     if err != nil {
77         utils.Logger().Error().
78             Err(err).
79             Msg("HandleMessage get message type failed")
80     }
81     return
82
83     msgPayload, err := proto.GetMessagePayload(content)
84     if err != nil {
85         utils.Logger().Error().
86             Err(err).
87             Msg("HandleMessage get message payload failed")
88     }
89     return
90
91     switch msgCategory {
92     case proto.Consensus:
93         msgPayload, _ := proto.GetConsensusMessagePayload(content)
94         if node.NodeConfig.Role() == nodeconfig.ExplorerNode {
95             node.ExplorerMessageHandler(msgPayload)
96         } else {
97             node.ConsensusMessageHandler(msgPayload)
98         }
99     case proto.DRand:
100         msgPayload, _ := proto.GetDRandMessagePayload(content)
101         if node.DRand != nil {
102             if node.DRand.IsLeader {
103                 node.DRand.ProcessMessageLeader(msgPayload)
104             } else {
105                 node.DRand.ProcessMessageValidator(msgPayload)
106             }
107         }
108     case proto.Node:
109         actionType := proto_node.MessageType(msgType)
110         switch actionType {
111         case proto_node.Transaction:
112             utils.Logger().Debug().Msg("NET: received message: Node/Transaction")
113             node.transactionMessageHandler(msgPayload)
114         case proto_node.Staking:

```

```

115         utils.Logger().Debug().Msg("NET: received message: Node/Staking")
116         node.stakingMessageHandler(msgPayload)
117         case proto_node.Block:
118             utils.Logger().Debug().Msg("NET: received message: Node/Block")
119             blockMsgType := proto_node.BlockMessageType(msgPayload[0])

```

Listing 3.13: node/node_handler.go

msgCategory, msgType, msgPayload are extracted from the message (msg[0], msg[1], msg[2:]), and HandleMessage will take different actions according to them.

```

190 func (node *Node) stakingMessageHandler(msgPayload []byte) {
191     txs := staking.StakingTransactions{}
192     err := rlp.Decode(bytes.NewReader(msgPayload[:]), &txs)
193     if err != nil {
194         utils.Logger().Error().
195             Err(err).
196             Msg("Failed to deserialize staking transaction list")
197         return
198     }
199     node.addPendingStakingTransactions(txs)
200 }

```

Listing 3.14: node/node_handler.go

stakingMessageHandler will be called for staking transaction messages. It will decode the staking transactions encoded in RLP format, and pass them to addPendingStakingTransactions (line 199).

```

295 func (node *Node) addPendingStakingTransactions(newStakingTxs staking.
    StakingTransactions) {
296     txPoolLimit := core.ShardingSchedule.MaxTxPoolSizeLimit()
297     node.pendingStakingTxMutex.Lock()
298     for _, tx := range newStakingTxs {
299         if _, ok := node.pendingStakingTransactions[tx.Hash()]; !ok {
300             node.pendingStakingTransactions[tx.Hash()] = tx
301         }
302         if len(node.pendingStakingTransactions) > txPoolLimit {
303             break
304         }
305     }
306     node.pendingStakingTxMutex.Unlock()
307     ...
308 }

```

Listing 3.15: node/node.go

addPendingStakingTransactions will check whether the staking transaction had been recorded in the pendingStakingTransactions map (line 299), and will stop storing new transactions if the map size > txPoolLimit (8,000 in mainnet).

However, the length check is misplaced, it should be executed before storing the staking transaction into pendingStakingTransactions.

Specifically, a malicious attacker can flood the victims with many different staking transactions and gradually increase the memory usage of the `pendingStakingTransactions` map, which eventually could lead to resource exhausting and hang or crash the remote nodes in the end.

Recommendation Put the length check in the right place.

3.7 DoS Vulnerability in the P2P Module - #2

- ID: PVE-007
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `node/node.go`
- Category: Coding Practices [14]
- CWE subcategory: CWE-20 [15]

Description

This is a vulnerability in the P2P module, which could be exploited by attackers to perform DoS attack against the harmony network.

Within the harmony network, a node can be one of the these roles: validator, leader, beacon validator, or beacon leader depending on its context. With each role, a node would run a certain set of services.

Furthermore, harmony network has enabled libp2p based gossiping using pubsub. Nodes no longer send messages to individual nodes, instead, they publish / subscribe to different topics.

```

66 func (node *Node) HandleMessage(content []byte, sender libp2p_peer.ID) {
67     msgCategory, err := proto.GetMessageCategory(content)
68     if err != nil {
69         utils.Logger().Error().
70             Err(err).
71             Msg("HandleMessage get message category failed")
72     }
73     return
74
75     msgType, err := proto.GetMessageType(content)
76     if err != nil {
77         utils.Logger().Error().
78             Err(err).
79             Msg("HandleMessage get message type failed")
80     }
81     return
82
83     msgPayload, err := proto.GetMessagePayload(content)
84     if err != nil {
85         utils.Logger().Error().
86             Err(err).

```

```

87         Msg("HandleMessage get message payload failed")
88     return
89 }
90
91 switch msgCategory {
92 case proto.Consensus:
93     msgPayload, _ := proto.GetConsensusMessagePayload(content)
94     if node.NodeConfig.Role() == nodeconfig.ExplorerNode {
95         node.ExplorerMessageHandler(msgPayload)
96     } else {
97         node.ConsensusMessageHandler(msgPayload)
98     }
99 case proto.DRand:
100     msgPayload, _ := proto.GetDRandMessagePayload(content)
101     if node.DRand != nil {
102         if node.DRand.IsLeader {
103             node.DRand.ProcessMessageLeader(msgPayload)
104         } else {
105             node.DRand.ProcessMessageValidator(msgPayload)
106         }
107     }
108 case proto.Node:
109     actionType := proto_node.MessageType(msgType)
110     switch actionType {
111     case proto_node.Transaction:
112         utils.Logger().Debug().Msg("NET: received message: Node/Transaction")
113         node.transactionMessageHandler(msgPayload)
114     case proto_node.Staking:
115         utils.Logger().Debug().Msg("NET: received message: Node/Staking")
116         node.stakingMessageHandler(msgPayload)
117     case proto_node.Block:
118         utils.Logger().Debug().Msg("NET: received message: Node/Block")
119         blockMsgType := proto_node.BlockMessageType(msgPayload[0])
120         switch blockMsgType {
121         case proto_node.Sync:
122             utils.Logger().Debug().Msg("NET: received message: Node/Sync")
123             var blocks []*types.Block
124             err := rlp.DecodeBytes(msgPayload[1:], &blocks)
125             if err != nil {
126                 utils.Logger().Error().
127                     Err(err).
128                     Msg("block sync")
129             } else {
130                 // for non-beaconchain node, subscribe to beacon block broadcast
131                 if node.Blockchain().ShardID() != 0 {
132                     for _, block := range blocks {
133                         if block.ShardID() == 0 {
134                             utils.Logger().Info().
135                                 Uint64("block", block.NumberU64()).
136                                 Msgf("Block being handled by block channel %d %d",
137                                     block.NumberU64(), block.ShardID())
137                             node.BeaconBlockChannel <- block

```

```

138         }
139     }
140 }
141 if node.Client != nil && node.Client.UpdateBlocks != nil && blocks
    != nil {
142     utils.Logger().Info().Msg("Block being handled by client")
143     node.Client.UpdateBlocks(blocks)
144 }
145 }
146
147 case proto_node.Header:
148     // only beacon chain will accept the header from other shards
149     utils.Logger().Debug().Uint32("shardID", node.NodeConfig.ShardID).Msg("
    NET: received message: Node/Header")
150     if node.NodeConfig.ShardID != 0 {
151         return
152     }
153     node.ProcessHeaderMessage(msgPayload[1:]) // skip first byte which is
        blockMsgType
154
155 case proto_node.Receipt:
156     utils.Logger().Debug().Msg("NET: received message: Node/Receipt")
157     node.ProcessReceiptMessage(msgPayload[1:]) // skip first byte which is
        blockMsgType
158
159 }

```

Listing 3.16: node/node_handler.go

msgCategory, msgType, msgPayload are extracted from the message (msg[0], msg[1], msg[2:]), and HandleMessage will take different actions according to them.

```

406 func (node *Node) ProcessReceiptMessage(msgPayload []byte) {
407     cxp := types.CXReceiptsProof{}
408     if err := rlp.DecodeBytes(msgPayload, &cxp); err != nil {
409         utils.Logger().Error().Err(err).Msg("[ProcessReceiptMessage] Unable to Decode
            message Payload")
410         return
411     }
412     utils.Logger().Debug().Interface("cxp", cxp).Msg("[ProcessReceiptMessage] Add
        CXReceiptsProof to pending Receipts")
413     // TODO: integrate with txpool
414     node.AddPendingReceipts(&cxp)
415 }

```

Listing 3.17: node/node_cross_shard.go

ProcessReceiptMessage would be called for receipts messages. It would decode the cross shard receipts and merkle proof encoded in RLP format, and pass them to AddPendingReceipts (line 414).

```

329 func (node *Node) AddPendingReceipts(receipts *types.CXReceiptsProof) {
330     node.pendingCXMutex.Lock()
331     defer node.pendingCXMutex.Unlock()

```



```

332
333     if receipts.ContainsEmptyField() {
334         utils.Logger().Info().Int("totalPendingReceipts", len(node.pendingCXReceipts)).
335             Msg("CXReceiptsProof contains empty field")
336     }
337
338     blockNum := receipts.Header.Number().Uint64()
339     shardID := receipts.Header.ShardID()
340     key := utils.GetPendingCXKey(shardID, blockNum)
341
342     if _, ok := node.pendingCXReceipts[key]; ok {
343         utils.Logger().Info().Int("totalPendingReceipts", len(node.pendingCXReceipts)).
344             Msg("Already Got Same Receipt message")
345     }
346     node.pendingCXReceipts[key] = receipts
347     utils.Logger().Info().Int("totalPendingReceipts", len(node.pendingCXReceipts)).Msg("
348         Got ONE more receipt message")

```

Listing 3.18: node/node.go

AddPendingReceipts would check whether the receipt had been recorded in the pendingCXReceipts map (line 342), and would save it if not (line 346).

However, there is no limitation enforced while adding new receipts into pendingCXReceipts.

Therefore, a malicious attacker can flood the victims with many crafted receipts and gradually increase the memory usage of the pendingCXReceipts map, which eventually could lead to resource exhausting and hang or crash the remote nodes in the end.

Recommendation Add length limitation on the cross shard receipts.

3.8 Integer Overflow in the RPC Module

- ID: PVE-008
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: [internal/hmyapi/transactionpool.go](#)
- Category: Coding Practices [14]
- CWE subcategory: CWE-190 [20]

Description

This is a vulnerability in the RPC api GetTransactionsHistory, which could be exploited by attackers to perform DoS attack against RPC thread.

```

45 // GetTransactionsHistory returns the list of transactions hashes that involve a
    particular address.
46 func (s *PublicTransactionPoolAPI) GetTransactionsHistory(ctx context.Context, args
    TxHistoryArgs) (map[string]interface{}, error) {
47     address := args.Address
48     result := []common.Hash{}
49     var err error
50     if strings.HasPrefix(args.Address, "one1") {
51         address = args.Address
52     } else {
53         addr := internal_common.ParseAddr(args.Address)
54         address, err = internal_common.AddressToBech32(addr)
55         if err != nil {
56             return nil, err
57         }
58     }
59     hashes, err := s.b.GetTransactionsHistory(address, args.TxType, args.Order)
60     if err != nil {
61         return nil, err
62     }
63     result = ReturnWithPagination(hashes, args)

```

Listing 3.19: internal/hmyapi/transactionpool.go

When analyzing the above code snippet, we noticed that harmony network allows a user to request transactions hashes history (line 59) by passing parameters of TxHistoryArgs struct:

```

24 // TxHistoryArgs is struct to make GetTransactionsHistory request
25 type TxHistoryArgs struct {
26     Address    string `json:"address"`
27     PageIndex  int    `json:"pageIndex"`
28     PageSize   int    `json:"pageSize"`
29     FullTx     bool   `json:"fullTx"`
30     TxType     string `json:"txType"`
31     Order      string `json:"order"`
32 }

```

Listing 3.20: internal/hmyapi/transactionpool.go

The ReturnWithPagination routine use two parameters PageIndex/pageSize to return transactions history with pagination:

```

19 // ReturnWithPagination returns result with tran (offset, page in TxHistoryArgs).
20 func ReturnWithPagination(hashes []common.Hash, args TxHistoryArgs) []common.Hash {
21     pageSize := defaultPageSize
22     pageIndex := args.PageIndex
23     if args.PageSize > 0 {
24         pageSize = args.PageSize
25     }
26     if pageSize*pageIndex >= len(hashes) {
27         return make([]common.Hash, 0)
28     }
29     if pageSize*pageIndex+pageSize > len(hashes) {

```

```

30     return hashes[pageSize*pageIndex:]
31 }
32 return hashes[pageSize*pageIndex : pageSize*pageIndex+pageSize]
33 }

```

Listing 3.21: internal/hmyapi/util.go

However, these parameters are directly passed from a user-controlled transaction and thus they should be validated before usage. Although in current implementation, such validation is insufficient and malicious parameters, i.e., pageIndex and pageSize, can cause an array OOB Panic (lines 32).

Recommendation Add sanity checks for these parameters.

3.9 Consensus Suspending in the Consensus Module - #1

- ID: PVE-009
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: consensus/checks.go
- Category: Behavioral Problems [16]
- CWE subcategory: CWE-841 [17]

Description

This is a vulnerability in the consensus module, which could be exploited by attackers to compromise the harmony network consensus. As an improvement on PBFT, Harmony's consensus protocol is linearly scalable in terms of communication complexity, and thus it is called Fast Byzantine Fault Tolerance (FBFT). Specifically, Harmony's FBFT consensus involves the following steps as shown in Figure 3.1:

The first phase is announce, the leader broadcasts announce message (e.g. the proposal block) to validators. When a validator receives announce message, it enters prepare phase.

```

17 func (consensus *Consensus) onAnnounce(msg *msg_pb.Message) {
18     recvMsg, err := ParseFBFTMessage(msg)
19     if err != nil {
20         consensus.getLogger().Error().
21             Err(err).
22             Uint64("MsgBlockNum", recvMsg.BlockNum).
23             Msg("[OnAnnounce] Unparseable leader message")
24         return
25     }
26
27     // NOTE let it handle its own logs
28     if !consensus.onAnnounceSanityChecks(recvMsg) {
29         return

```

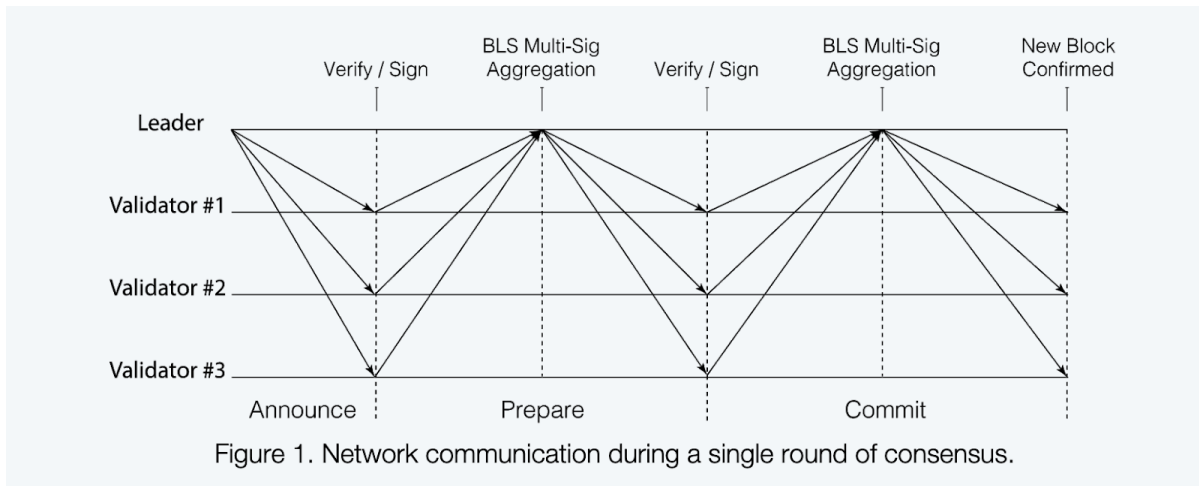


Figure 3.1: FBFT Consensus

```
30     }
```

Listing 3.22: consensus/validator.go

`onAnnounce` is called when validators receives announce message from the leader. It performs lots of sanity check to make sure the message is valid.

```
91 func (consensus *Consensus) onAnnounceSanityChecks(recvMsg *FBFTMessage) bool {
92     logMsgs := consensus.FBFTLog.GetMessagesByTypeSeqView(
93         msg_pb.MessageType_ANNOUNCE, recvMsg.BlockNum, recvMsg.ViewID,
94     )
95     if len(logMsgs) > 0 {
96         if logMsgs[0].BlockHash != recvMsg.BlockHash &&
97             logMsgs[0].SenderPubkey.IsEqual(recvMsg.SenderPubkey) {
98             consensus.getLogger().Debug().
99                 Str("logMsgSenderKey", logMsgs[0].SenderPubkey.SerializeToHexStr()).
100                 Str("logMsgBlockHash", logMsgs[0].BlockHash.Hex()).
101                 Str("recvMsg.SenderPubkey", recvMsg.SenderPubkey.SerializeToHexStr()).
102                 Uint64("recvMsg.BlockNum", recvMsg.BlockNum).
103                 Uint64("recvMsg.ViewID", recvMsg.ViewID).
104                 Str("recvMsgBlockHash", recvMsg.BlockHash.Hex()).
105                 Str("LeaderKey", consensus.LeaderPubKey.SerializeToHexStr()).
106                 Msg("[OnAnnounce] Leader is malicious")
107             if consensus.current.Mode() == ViewChanging {
108                 viewID := consensus.current.ViewID()
109                 consensus.startViewChange(viewID + 1)
110             } else {
111                 consensus.startViewChange(consensus.viewID + 1)
112             }
113         }
114     }
115 }
```

Listing 3.23: consensus/checks.go

When a validator detects the leader proposed two different announce messages in one view, it

would immediately start view change (line 107-112). However, the sanity check in `onAnnounce` can't guarantee the message is from the current leader.

```

9 func (consensus *Consensus) validatorSanityChecks(msg *msg_pb.Message) bool {
10     senderKey, err := consensus.verifySenderKey(msg)
11     if err != nil {
12         if err == errValidNotInCommittee {
13             consensus.getLogger().Info().
14                 Msg("sender key not in this slot's subcommittee")
15         } else {
16             consensus.getLogger().Error().Err(err).Msg("VerifySenderKey failed")
17         }
18         return false
19     }
20
21     if !senderKey.IsEqual(consensus.LeaderPubKey) &&
22         consensus.current.Mode() == Normal && !consensus.ignoreViewIDCheck {
23         consensus.getLogger().Warn().Msg("[OnPrepared] SenderKey not match leader PubKey")
24         return false
25     }
26
27     if err := verifyMessageSig(senderKey, msg); err != nil {
28         consensus.getLogger().Error().Err(err).Msg(
29             "Failed to verify sender's signature",
30         )
31         return false
32     }
33
34     return true
35 }

```

Listing 3.24: `consensus/checks.go`

Specifically, a malicious leader can intentionally propose two different announce messages in one view to trigger validators' view change and make them all transit to `ViewChanging` mode. Once validators are in `ViewChanging` mode, the sanity checks (line 21-23) are ignored, thus the leader can constantly trigger view change by sending different announce messages with the same block number and view id, eventually compromise the whole harmony network.

Recommendation Add sanity checks for the legality of the announce messages.

3.10 Out-of-Memory in the Consensus Module - #1

- ID: PVE-010
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `consensus/checks.go`
- Category: Behavioral Problems [16]
- CWE subcategory: CWE-115 [21]

Description

Under harmony's PBFT consensus protocol, the first phase is announce, the leader would broadcast announce message (e.g. the proposal block) to validators. When a validator receives announce message, it enters prepare phase.

```

17 func (consensus *Consensus) onAnnounce(msg *msg_pb.Message) {
18     recvMsg, err := ParseFBFTMessage(msg)
19     if err != nil {
20         consensus.getLogger().Error().
21             Err(err).
22             Uint64("MsgBlockNum", recvMsg.BlockNum).
23             Msg("[OnAnnounce] Unparseable leader message")
24         return
25     }
26
27     // NOTE let it handle its own logs
28     if !consensus.onAnnounceSanityChecks(recvMsg) {
29         return
30     }
31
32     consensus.getLogger().Debug().
33         Uint64("MsgViewID", recvMsg.ViewID).
34         Uint64("MsgBlockNum", recvMsg.BlockNum).
35         Msg("[OnAnnounce] Announce message Added")
36     consensus.FBFTLog.AddMessage(recvMsg)

```

Listing 3.25: consensus/validator.go

`onAnnounce` is called when validators receives announce message from the leader. It performs lots of sanity check to make sure the message is valid. It would store the message for future validation (line 36).

```

91 func (consensus *Consensus) onAnnounceSanityChecks(recvMsg *FBFTMessage) bool {
92     logMsgs := consensus.FBFTLog.GetMessagesByTypeSeqView(
93         msg_pb.MessageType_ANNOUNCE, recvMsg.BlockNum, recvMsg.ViewID,
94     )
95     if len(logMsgs) > 0 {
96         if logMsgs[0].BlockHash != recvMsg.BlockHash &&
97             logMsgs[0].SenderPubkey.IsEqual(recvMsg.SenderPubkey) {
98             consensus.getLogger().Debug().
99                 Str("logMsgSenderKey", logMsgs[0].SenderPubkey.SerializeToHexStr()).
100                 Str("logMsgBlockHash", logMsgs[0].BlockHash.Hex()).
101                 Str("recvMsg.SenderPubkey", recvMsg.SenderPubkey.SerializeToHexStr()).
102                 Uint64("recvMsg.BlockNum", recvMsg.BlockNum).
103                 Uint64("recvMsg.ViewID", recvMsg.ViewID).
104                 Str("recvMsgBlockHash", recvMsg.BlockHash.Hex()).
105                 Str("LeaderKey", consensus.LeaderPubKey.SerializeToHexStr()).
106                 Msg("[OnAnnounce] Leader is malicious")
107             if consensus.current.Mode() == ViewChanging {
108                 viewID := consensus.current.ViewID()
109                 consensus.startViewChange(viewID + 1)

```

```

110         } else {
111             consensus.startViewChange(consensus.viewID + 1)
112         }
113     }
114     consensus.getLogger().Debug().
115         Str("leaderKey", consensus.LeaderPubKey.SerializeToHexStr()).
116         Msg("[OnAnnounce] Announce message received again")
117 }
118 return consensus.isRightBlockNumCheck(recvMsg)
119 }
120
121 func (consensus *Consensus) isRightBlockNumCheck(recvMsg *FBFTMessage) bool {
122     if recvMsg.BlockNum < consensus.blockNum {
123         consensus.getLogger().Debug().
124             Uint64("MsgBlockNum", recvMsg.BlockNum).
125             Msg("Wrong BlockNum Received, ignoring!")
126         return false
127     }
128     return true
129 }

```

Listing 3.26: consensus/checks.go

When a validator detects the leader proposed two different announce messages in one view, it would immediately start view change (line 107-112). However, the sanity check in `onAnnounce` can't guarantee the message is from the current leader.

```

9 func (consensus *Consensus) validatorSanityChecks(msg *msg_pb.Message) bool {
10     senderKey, err := consensus.verifySenderKey(msg)
11     if err != nil {
12         if err == errValidNotInCommittee {
13             consensus.getLogger().Info().
14                 Msg("sender key not in this slot's subcommittee")
15         } else {
16             consensus.getLogger().Error().Err(err).Msg("VerifySenderKey failed")
17         }
18         return false
19     }
20
21     if !senderKey.IsEqual(consensus.LeaderPubKey) &&
22         consensus.current.Mode() == Normal && !consensus.ignoreViewIDCheck {
23         consensus.getLogger().Warn().Msg("[OnPrepared] SenderKey not match leader PubKey")
24         return false
25     }
26
27     if err := verifyMessageSig(senderKey, msg); err != nil {
28         consensus.getLogger().Error().Err(err).Msg(
29             "Failed to verify sender's signature",
30         )
31         return false
32     }
33 }

```

```

34     return true
35 }

```

Listing 3.27: consensus/checks.go

Therefore, a malicious leader could intentionally propose two different announce messages in one view to trigger validators' view change (line 107-112) and make them all transit to ViewChanging mode. Once validators are in ViewChanging mode, the sanity checks (line 21-25) are ignored, and the leader can flood these validators by sending lots of announce messages with large block number (line 122) so the validators would keep storing these messages and eventually cause them out of memory.

Recommendation Add sanity checks for the legality of the announce messages.

3.11 Out-of-Memory in the Consensus Module - #2

- ID: PVE-011
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: consensus/view_change.go
- Category: Behavioral Problems [16]
- CWE subcategory: CWE-115 [21]

Description

Under harmony's PBFT consensus protocol, there are two causes for validators to start view change process. One is when a validator detects the leader proposed two different announce messages in one view, it would immediately start view change. The other is a validator doesn't make any progress after timeout. There are two kinds of timeouts: timeout in normal consensus mode and timeout in view change mode.

```

112 func (consensus *Consensus) startViewChange(viewID uint64) {
113     if consensus.disableViewChange {
114         return
115     }
116     consensus.consensusTimeout[timeoutConsensus].Stop()
117     consensus.consensusTimeout[timeoutBootstrap].Stop()
118     consensus.current.SetMode(ViewChanging)
119     consensus.current.SetViewID(viewID)
120     consensus.LeaderPubKey = consensus.GetNextLeaderKey()
121
122     diff := viewID - consensus.viewID
123     duration := time.Duration(int64(diff) * int64(viewChangeDuration))
124     consensus.getLogger().Info().
125         Uint64("ViewChangingID", viewID).
126         Dur("timeoutDuration", duration).

```



```

127     Str("NextLeader", consensus.LeaderPubKey.SerializeToHexStr()).
128     Msg("[startViewChange]")
129
130     msgToSend := consensus.constructViewChangeMessage()
131     consensus.host.SendMessageToGroups([] nodeconfig.GroupID{
132         nodeconfig.NewGroupIDByShardID(nodeconfig.ShardID(consensus.ShardID)),
133     },
134         host.ConstructP2pMessage(byte(17), msgToSend),
135     )
136
137     consensus.consensusTimeout[timeoutViewChange].SetDuration(duration)
138     consensus.consensusTimeout[timeoutViewChange].Start()
139     consensus.getLogger().Debug().
140         Uint64("ViewChangingID", consensus.current.ViewID()).
141         Msg("[startViewChange] start view change timer")
142 }

```

Listing 3.28: consensus/view_change.go

startViewChange is called when a validator want to start a view change process. It would stop the consensus timer (line 116), set current mode to ViewChanging (line 118), construct and send out the view change message (line 130-135), and lastly, start the view change timer (line 137-138)

```

144 func (consensus *Consensus) onViewChange(msg *msg_pb.Message) {
145     recvMsg, err := ParseViewChangeMessage(msg)
146     if err != nil {
147         consensus.getLogger().Warn().Msg("[onViewChange] Unable To Parse Viewchange Message")
148     }
149     return
150 }
151 newLeaderKey := recvMsg.LeaderPubkey
152 if !consensus.PubKey.IsEqual(newLeaderKey) {
153     return
154 }
155 if consensus.Decider.IsQuorumAchieved(quorum.ViewChange) {
156     consensus.getLogger().Debug().
157         Int64("have", consensus.Decider.SignersCount(quorum.ViewChange)).
158         Int64("need", consensus.Decider.TwoThirdsSignersCount()).
159         Str("validatorPubKey", recvMsg.SenderPubkey.SerializeToHexStr()).
160         Msg("[onViewChange] Received Enough View Change Messages")
161     return
162 }
163
164 senderKey, err := consensus.verifyViewChangeSenderKey(msg)
165 if err != nil {
166     consensus.getLogger().Debug().Err(err).Msg("[onViewChange] VerifySenderKey Failed")
167     return
168 }
169
170 // TODO: if difference is only one, new leader can still propose the same committed
171 // block to avoid another view change

```

```

171 // TODO: new leader catchup without ignore view change message
172 if consensus.blockNum > recvMsg.BlockNum {
173     consensus.getLogger().Debug().
174         Uint64("MsgBlockNum", recvMsg.BlockNum).
175         Msg("[onViewChange] Message BlockNum Is Low")
176     return
177 }
178
179 if consensus.blockNum < recvMsg.BlockNum {
180     consensus.getLogger().Warn().
181         Uint64("MsgBlockNum", recvMsg.BlockNum).
182         Msg("[onViewChange] New Leader Has Lower Blocknum")
183     return
184 }
185
186 if consensus.current.Mode() == ViewChanging &&
187     consensus.current.ViewID() > recvMsg.ViewID {
188     consensus.getLogger().Warn().
189         Uint64("MyViewChangingID", consensus.current.ViewID()).
190         Uint64("MsgViewChangingID", recvMsg.ViewID).
191         Msg("[onViewChange] ViewChanging ID Is Low")
192     return
193 }
194 if err = verifyMessageSig(senderKey, msg); err != nil {
195     consensus.getLogger().Debug().Err(err).Msg("[onViewChange] Failed To Verify Sender's
196         Signature")
197     return
198 }
199 consensus.vcLock.Lock()
200 defer consensus.vcLock.Unlock()
201
202 // update the dictionary key if the viewID is first time received
203 consensus.addViewIDKeyIfNotExist(recvMsg.ViewID)

```

Listing 3.29: consensus/view_change.go

```

349 // received enough view change messages, change state to normal consensus
350 if consensus.Decider.IsQuorumAchievedByMask(consensus.viewIDBitmap[recvMsg.ViewID],
351     true) {
352     consensus.current.SetMode(Normal)
353     consensus.LeaderPubKey = consensus.PubKey
354     consensus.ResetState()
355     if len(consensus.m1Payload) == 0 {
356         // TODO(Chao): explain why ReadySignal is sent only in this case but not the
357             other case.
358         go func() {
359             consensus.ReadySignal <- struct{}{}
360         }()
361     } else {

```

Listing 3.30: consensus/view_change.go

```

395     consensus.current.setviewid(recvmsg.viewid)
396     msgtosend := consensus.constructnewviewmessage(recvmsg.viewid)
397
398     consensus.getLogger().Warn().
399     Int("payloadSize", len(consensus.m1Payload)).
400     Hex("M1Payload", consensus.m1Payload).
401     Msg("[onViewChange] Sent NewView Message")
402     consensus.msgSender.SendWithRetry(consensus.blockNum, msg_pb.MessageType_NEWVIEW, []
        nodeconfig.GroupID{nodeconfig.NewGroupIDByShardID(nodeconfig.ShardID(consensus.
        ShardID))}, host.ConstructP2pMessage(byte(17), msgToSend))
403
404     consensus.viewid = recvmsg.viewid
405     consensus.resetviewchangestate()
406     consensus.consensustimeout[timeoutviewchange].stop()
407     consensus.consensustimeout[timeoutconsensus].start()

```

Listing 3.31: consensus/view_change.go

onViewChange is responsible for view change handling for validators. First, make sure it's the next leader (line 151), then update the dictionary key if the viewID is received for the first time (line 203)

Once new leader receives enough view change messages (line 350), it would change state to normal, and reset the consensus state (line 351-353). Finally, new leader will construct and send out a new view message to others (line 396-402), and reset its view change state (line 405). However, there is no constraint on updating the dictionary key.

```

628 func (consensus *Consensus) addViewIDKeyIfNotExist(viewID uint64) {
629     members := consensus.Decider.Participants()
630     if _, ok := consensus.bhpSigs[viewID]; !ok {
631         consensus.bhpSigs[viewID] = map[string]*bls.Sign{}
632     }
633     if _, ok := consensus.nilSigs[viewID]; !ok {
634         consensus.nilSigs[viewID] = map[string]*bls.Sign{}
635     }
636     if _, ok := consensus.viewIDSigs[viewID]; !ok {
637         consensus.viewIDSigs[viewID] = map[string]*bls.Sign{}
638     }
639     if _, ok := consensus.bhpBitmap[viewID]; !ok {
640         bhpBitmap, _ := bls_cosi.NewMask(members, nil)
641         consensus.bhpBitmap[viewID] = bhpBitmap
642     }
643     if _, ok := consensus.nilBitmap[viewID]; !ok {
644         nilBitmap, _ := bls_cosi.NewMask(members, nil)
645         consensus.nilBitmap[viewID] = nilBitmap
646     }
647     if _, ok := consensus.viewIDBitmap[viewID]; !ok {
648         viewIDBitmap, _ := bls_cosi.NewMask(members, nil)
649         consensus.viewIDBitmap[viewID] = viewIDBitmap
650     }
651 }

```

Listing 3.32: consensus/consensus_service.go

A malicious validator could flood next leader by sending lots of view change messages with different viewID. `addViewIDKeyIfNotExist` would make new maps and masks for first time received new viewID. On the other hand, the crafted view change messages may never achieve quorum to trigger view change process and clear the view state, so in the end, the next leader would run out of memory.

Recommendation Add sanity checks for the viewID of view change messages.

3.12 Consensus Suspending in the Consensus Module - #2

- ID: PVE-012
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `consensus/view_change.go`
- Category: Behavioral Problems [16]
- CWE subcategory: CWE-115 [21]

Description

Under harmony's PBFT consensus protocol, there are two causes for validators to start view change process. One is when a validator detects the leader proposed two different announce messages in one view, it would immediately start view change. The other is a validator doesn't make any progress after timeout. There are two kinds of timeouts: timeout in normal consensus mode and timeout in view change mode.

The view change process is as follows:

- 1) When the consensus timer timeouts, a node starts view change by sending view change message including viewID and prepared message (containing $\geq 2f+1$ aggregated signatures) to new leader. If it doesn't receive prepared message, it just sends view change message including signature on viewID but without prepared message.
- 2) When the new leader receives enough ($\geq 2f+1$) view change messages, it aggregates signatures of viewID and just pick one prepared message from view change messages. It broadcasts new view message including aggregated signatures as well as the picked prepared message. Then the new leader switches to normal consensus mode. A validator switches to normal consensus node when it receives new view message from the new leader, at the same time, it stops the view change timer and start the consensus timer. If the validator doesn't receive new view message before view change timeout, it would increase viewID by one and start another view change.

```

440 if recvMsg.M3AggSig == nil || recvMsg.M3Bitmap == nil {
441     consensus.getLogger().Error().Msgf("[onNewView] M3AggSig or M3Bitmap is nil")
442     return
443 }
444 m3Sig := recvMsg.M3AggSig
445 m3Mask := recvMsg.M3Bitmap
446
447 viewIDBytes := make([]byte, 8)
448 binary.LittleEndian.PutUint64(viewIDBytes, recvMsg.ViewID)
449
450 if !consensus.Decider.IsQuorumAchievedByMask(m3Mask, true) {
451     consensus.getLogger().Warn().
452         Msgf("[onNewView] Quorum Not achieved")
453     return
454 }
455
456 if !m3Sig.VerifyHash(m3Mask.AggregatePublic, viewIDBytes) {
457     consensus.getLogger().Warn().
458         Str("m3Sig", m3Sig.SerializeToHexStr()).
459         Hex("m3Mask", m3Mask.Bitmap).
460         Uint64("MsgViewID", recvMsg.ViewID).
461         Msgf("[onNewView] Unable to Verify Aggregated Signature of M3 (ViewID) payload")
462     return
463 }

```

Listing 3.33: consensus/view_change.go

```

515 // newView message verified success, override my state
516 consensus.viewID = recvMsg.ViewID
517 consensus.current.SetViewID(recvMsg.ViewID)
518 consensus.LeaderPubKey = senderKey
519 consensus.ResetViewChangeState()
520
521 // change view and leaderKey to keep in sync with network
522 if consensus.blockNum != recvMsg.BlockNum {
523     consensus.getLogger().Debug().

```

Listing 3.34: consensus/view_change.go

`onNewView` is called when a validator receives the new view message from new leader at step 2. It would check whether the signature of viewID is valid and achieved the quorum (line 440-463). If verified successfully, the consensus state would be updated (line 516-519).

However, some sanity checks are missing:

- 1) `recvMsg.ViewID` should `>` `consensus.current.ViewID()`
- 2) new view message should only come from next leader

A malicious next leader could save the new view message, and broadcast out whenever it wants to become leader, thus break the harmony network consensus. On the other hand, any committee

member can also send the new view message to other validators, though it may not become the leader, can still compromise the consensus.

Recommendation Add sanity checks for the viewID of new view messages.

3.13 Consensus Suspending in the Consensus Module - #3

- ID: PVE-013
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: consensus/checks.go
- Category: Input Validation Issues [22]
- CWE subcategory: CWE-349 [23]

Description

This is a vulnerability in the consensus module, which could be exploited by attackers to compromise the harmony network consensus.

As an improvement on PBFT, Harmony's consensus protocol is linearly scalable in terms of communication complexity, and thus it is called Fast Byzantine Fault Tolerance (FBFT).

In prepare phase, the validator sends prepare message (e.g. signature on blockhash) to leader. When leader receives enough (i.e. $\geq 2f+1$) prepare messages, it aggregates signatures of prepare messages received from validators and sends out prepared message contains aggregated prepare signatures and the candidate block.

```

94 func (consensus *Consensus) onPrepared(msg *msg_pb.Message) {
95     recvMsg, err := ParseFBFTMessage(msg)
96     if err != nil {
97         consensus.getLogger().Debug().Err(err).Msg("[OnPrepared] Unparseable validator
           message")
98         return
99     }
100    consensus.getLogger().Info().
101        Uint64("MsgBlockNum", recvMsg.BlockNum).
102        Uint64("MsgViewID", recvMsg.ViewID).
103        Msg("[OnPrepared] Received prepared message")
104
105    if recvMsg.BlockNum < consensus.blockNum {
106        consensus.getLogger().Debug().Uint64("MsgBlockNum", recvMsg.BlockNum).
107            Msg("Wrong BlockNum Received, ignoring!")
108        return
109    }
110
111    // check validity of prepared signature
112    blockHash := recvMsg.BlockHash
113    aggSig, mask, err := consensus.ReadSignatureBitmapPayload(recvMsg.Payload, 0)

```

```

114     if err != nil {
115         consensus.getLogger().Error().Err(err).Msg("ReadSignatureBitmapPayload failed!")
116         return
117     }
118
119     if !consensus.Decider.IsQuorumAchievedByMask(mask) {
120         consensus.getLogger().Warn().
121             Msgf("[OnPrepared] Quorum Not achieved")
122         return
123     }
124
125     if !aggSig.VerifyHash(mask.AggregatePublic, blockHash[:]) {
126         myBlockHash := common.Hash{}
127         myBlockHash.SetBytes(consensus.blockHash[:])
128         consensus.getLogger().Warn().
129             Uint64("MsgBlockNum", recvMsg.BlockNum).
130             Uint64("MsgViewID", recvMsg.ViewID).
131             Msgf("[OnPrepared] failed to verify multi signature for prepare phase")
132         return
133     }
134
135     // check validity of block
136     var blockObj types.Block
137     if err := rlp.DecodeBytes(recvMsg.Block, &blockObj); err != nil {
138         consensus.getLogger().Warn().
139             Err(err).
140             Uint64("MsgBlockNum", recvMsg.BlockNum).
141             Msgf("[OnPrepared] Unparseable block header data")
142         return
143     }
144     // let this handle it own logs
145     if !consensus.onPreparedSanityChecks(&blockObj, recvMsg) {
146         return
147     }
148     consensus.mutex.Lock()
149     defer consensus.mutex.Unlock()
150
151     consensus.FBFTLog.AddBlock(&blockObj)
152     .....

```

Listing 3.35: consensus/validator.go

onPrepared is called when validators receives prepared message from the leader. It would perform lots of sanity checks to make sure the message is valid. If it's legit, validators would store the attached block (line 151)

```

234 func (consensus *Consensus) onCommitted(msg *msg_pb.Message) {
235     recvMsg, err := ParseFBFTMessage(msg)
236     if err != nil {
237         consensus.getLogger().Warn().Msgf("[OnCommitted] unable to parse msg")
238         return
239     }

```

```

240 // NOTE let it handle its own logs
241 if !consensus.isRightBlockNumCheck(recvMsg) {
242     return
243 }
244
245 aggSig, mask, err := consensus.ReadSignatureBitmapPayload(recvMsg.Payload, 0)
246 if err != nil {
247     consensus.getLogger().Error().Err(err).Msg("[OnCommitted] readSignatureBitmapPayload
248         failed")
249     return
250 }
251
252 if !consensus.Decider.IsQuorumAchievedByMask(mask) {
253     consensus.getLogger().Warn().
254         Msgf("[OnCommitted] Quorum Not achieved")
255     return
256 }
257
258 // TODO(audit): verify signature on hash+blockNum+viewID (add a hard fork)
259 blockNumBytes := make([]byte, 8)
260 binary.LittleEndian.PutUint64(blockNumBytes, recvMsg.BlockNum)
261 commitPayload := append(blockNumBytes, recvMsg.BlockHash[:]...)
262 if !aggSig.VerifyHash(mask.AggregatePublic, commitPayload) {
263     consensus.getLogger().Error().
264         Uint64("MsgBlockNum", recvMsg.BlockNum).
265         Msg("[OnCommitted] Failed to verify the multi signature for commit phase")
266     return
267 }
268
269 consensus.FBFTLog.AddMessage(recvMsg)
270 consensus.ChainReader.WriteLastCommits(recvMsg.Payload)
271 consensus.getLogger().Debug().
272     Uint64("MsgViewID", recvMsg.ViewID).
273     Uint64("MsgBlockNum", recvMsg.BlockNum).
274     Msg("[OnCommitted] Committed message added")
275
276 consensus.mutex.Lock()
277 defer consensus.mutex.Unlock()
278
279 consensus.aggregatedCommitSig = aggSig
280 consensus.commitBitmap = mask
281
282 if recvMsg.BlockNum-consensus.blockNum > consensusBlockNumBuffer {
283     consensus.getLogger().Debug().Uint64("MsgBlockNum", recvMsg.BlockNum).Msg("[
284         OnCommitted] out of sync")
285     go func() {
286         select {
287         case consensus.BlockNumLowChan <- struct{}{}:
288             consensus.current.SetMode(Syncing)
289             for _, v := range consensus.consensusTimeout {
290                 v.Stop()
291             }

```



```

290     case <-time.After(1 * time.Second):
291     }
292     }()
293     return
294 }
295
296 consensus.tryCatchup()
297 if consensus.current.Mode() == ViewChanging {
298     consensus.getLogger().Debug().Msg("[OnCommitted] Still in ViewChanging mode, Exiting
299         !!")
300     return
301 }
302 if consensus.consensusTimeout[timeoutBootstrap].IsActive() {
303     consensus.consensusTimeout[timeoutBootstrap].Stop()
304     consensus.getLogger().Debug().Msg("[OnCommitted] Start consensus timer; stop
305         bootstrap timer only once")
306 } else {
307     consensus.getLogger().Debug().Msg("[OnCommitted] Start consensus timer")
308 }
309 consensus.consensusTimeout[timeoutConsensus].Start()
310 }

```

Listing 3.36: consensus/validator.go

After all validators agreed on the prepared message and enough commit messages are collected by the leader (i.e. $\geq 2f+1$), it would send committed message. `onCommitted` is called to handle the message, it would also perform lots of sanity checks to make sure the message is valid. If all seem right, this round is finished and the consensus timer would be reset (line 308).

However, the sanity checks in `onPreparedSanityChecks` can't guarantee the block is valid.

```

130 func (consensus *Consensus) onPreparedSanityChecks(
131     blockObj *types.Block, recvMsg *FBFTMessage,
132 ) bool {
133     if blockObj.NumberU64() != recvMsg.BlockNum ||
134         recvMsg.BlockNum < consensus.blockNum {
135         consensus.getLogger().Warn().
136             Uint64("MsgBlockNum", recvMsg.BlockNum).
137             Uint64("blockNum", blockObj.NumberU64()).
138             Msg("[OnPrepared] BlockNum not match")
139         return false
140     }
141     if blockObj.Header().Hash() != recvMsg.BlockHash {
142         consensus.getLogger().Warn().
143             Uint64("MsgBlockNum", recvMsg.BlockNum).
144             Hex("MsgBlockHash", recvMsg.BlockHash[:]).
145             Str("blockObjHash", blockObj.Header().Hash().Hex()).
146             Msg("[OnPrepared] BlockHash not match")
147         return false
148     }
149     if consensus.current.Mode() == Normal {
150         err := chain.Engine.VerifyHeader(consensus.ChainReader, blockObj.Header(), true)

```

```

151     if err != nil {
152         consensus.getLogger().Error().
153             Err(err).
154             Str("inChain", consensus.ChainReader.CurrentHeader().Number().String()).
155             Str("MsgBlockNum", blockObj.Header().Number().String()).
156             Msg("[OnPrepared] Block header is not verified successfully")
157         return false
158     }
159     if consensus.BlockVerifier == nil {
160         // do nothing
161     } else if err := consensus.BlockVerifier(blockObj); err != nil {
162         consensus.getLogger().Error().Err(err).Msg("[OnPrepared] Block verification failed")
163         return false
164     }
165 }
166
167 return true
168 }

```

Listing 3.37: consensus/checks.go

Specifically, a malicious leader can bypass the sanity checks (line 149-165) by making validators switch to Syncing mode.

```

205 func (node *Node) DoSyncing(bc *core.BlockChain, worker *worker.Worker,
206     willJoinConsensus bool) {
207     // TODO ek infinite loop; add shutdown/cleanup logic
208     SyncingLoop:
209     for {
210         if node.stateSync == nil {
211             node.stateSync = syncing.CreateStateSync(node.SelfPeer.IP, node.SelfPeer.Port,
212                 node.GetSyncID())
213             utils.Logger().Debug().Msg("[SYNC] initialized state sync")
214         }
215         if node.stateSync.GetActivePeerNumber() < MinConnectedPeers {
216             shardID := bc.ShardID()
217             peers, err := node.SyncingPeerProvider.SyncingPeers(shardID)
218             if err != nil {
219                 utils.Logger().Warn().
220                     Err(err).
221                     Uint32("shard_id", shardID).
222                     Msg("cannot retrieve syncing peers")
223                 continue SyncingLoop
224             }
225             if err := node.stateSync.CreateSyncConfig(peers, false); err != nil {
226                 utils.Logger().Warn().
227                     Err(err).
228                     Interface("peers", peers).
229                     Msg("[SYNC] create peers error")
230                 continue SyncingLoop
231             }
232         }
233     }

```

```

231     utils.Logger().Debug().Int("len", node.stateSync.GetActivePeerNumber()).Msg("[SYNC
        ] Get Active Peers")
232 }
233 // TODO: treat fake maximum height
234 if node.stateSync.IsOutOfSync(bc) {
235     node.stateMutex.Lock()
236     node.State = NodeNotInSync
237     node.stateMutex.Unlock()
238     if willJoinConsensus {
239         node.Consensus.BlocksNotSynchronized()
240     }
241     node.stateSync.SyncLoop(bc, worker, false, node.Consensus)
242     if willJoinConsensus {
243         node.stateMutex.Lock()
244         node.State = NodeReadyForConsensus
245         node.stateMutex.Unlock()
246         node.Consensus.BlocksSynchronized()
247     }
248 }
249 node.stateMutex.Lock()
250 node.State = NodeReadyForConsensus
251 node.stateMutex.Unlock()
252 // TODO on demand syncing
253 time.Sleep(time.Duration(node.syncFreq) * time.Second)
254 }
255 }

```

Listing 3.38: node/node_syncing.go

A node would switch to Syncing mode if the node thinks it's out of sync (line 234, 239). How a node decides whether it's out of sync is by asking other peers about their block height.

```

403 case downloader_pb.DownloaderRequest_BLOCKHEIGHT:
404     response.BlockHeight = node.Blockchain().CurrentBlock().NumberU64()
405 }

```

Listing 3.39: node/node_syncing.go

Theoretically, the malicious leader could trick other committee into Syncing mode by returning a fake high block height. Once validators are in Syncing mode, they would skip the block sanity checks in `onPreparedSanityChecks`, store whatever kind of block the leader sent and reply with the corresponding commit message. Once leader has enough commit message, it would send committed message to validators, and start a new round.

```

208 func (consensus *Consensus) tryCatchup() {
209     consensus.getLogger().Info().Msg("[TryCatchup] commit new blocks")
210     currentBlockNum := consensus.blockNum
211     for {
212         msgs := consensus.FBFTLog.GetMessagesByTypeSeq(msg_pb.MessageType_COMMITTED,
                consensus.blockNum)
213         if len(msgs) == 0 {
214             break

```

```

215 }
216 if len(msgs) > 1 {
217     consensus.getLogger().Error().
218         Int("numMsgs", len(msgs)).
219         Msg("[TryCatchup] DANGER!!! we should only get one committed message for a given
            blockNum")
220 }
221 consensus.getLogger().Info().Msg("[TryCatchup] committed message found")
222
223 block := consensus.FBFTLog.GetBlockByHash(msgs[0].BlockHash)
224 if block == nil {
225     break
226 }
227
228 if consensus.BlockVerifier == nil {
229     // do nothing
230 } else if err := consensus.BlockVerifier(block); err != nil {
231     consensus.getLogger().Info().Msg("[TryCatchup] block verification failed")
232     return
233 }
234
235 if block.ParentHash() != consensus.ChainReader.CurrentHeader().Hash() {
236     consensus.getLogger().Debug().Msg("[TryCatchup] parent block hash not match")
237     break
238 }
239 consensus.getLogger().Info().Msg("[TryCatchup] block found to commit")
240
241 preparedMsgs := consensus.FBFTLog.GetMessagesByTypeSeqHash(
242     msg_pb.MessageType_PREPARED, msgs[0].BlockNum, msgs[0].BlockHash,
243 )
244 msg := consensus.FBFTLog.FindMessageByMaxViewID(preparedMsgs)
245 if msg == nil {
246     break
247 }
248 consensus.getLogger().Info().Msg("[TryCatchup] prepared message found to commit")
249
250 // TODO(Chao): Explain the reasoning for these code
251 consensus.blockHash = [32]byte{}
252 consensus.blockNum = consensus.blockNum + 1
253 consensus.viewID = msgs[0].ViewID + 1
254 consensus.LeaderPubKey = msgs[0].SenderPubkey
255
256 consensus.getLogger().Info().Msg("[TryCatchup] Adding block to chain")
257 consensus.OnConsensusDone(block, msgs[0].Payload)
258 consensus.ResetState()
259
260 select {
261 case consensus.VerifiedNewBlock <- block:
262 default:
263     consensus.getLogger().Info().
264         Str("blockHash", block.Hash().String()).
265         Msg("[TryCatchup] consensus verified block send to chan failed")

```

```

266     continue
267 }
268
269 break
270 }
271 if currentBlockNum < consensus.blockNum {
272     consensus.getLogger().Info().
273         Uint64("From", currentBlockNum).
274         Uint64("To", consensus.blockNum).
275         Msg("[TryCatchup] Caught up!")
276     consensus.switchPhase(FBFTAnnounce, true)
277 }
278 // catup up and skip from view change trap
279 if currentBlockNum < consensus.blockNum &&
280     consensus.current.Mode() == ViewChanging {
281     consensus.current.SetMode(Normal)
282     consensus.consensusTimeout[timeoutViewChange].Stop()
283 }
284 // clean up old log
285 consensus.FBFTLog.DeleteBlocksLessThan(consensus.blockNum - 1)
286 consensus.FBFTLog.DeleteMessagesLessThan(consensus.blockNum - 1)
287 }
288 }

```

Listing 3.40: consensus/consensus_v2.go

However, this round may not be able to complete. Say, if the leader sends a malformed block within the prepared message and bypasses the block sanity checks as we explained above, then `tryCatchup` would not be able to proceed because the validity checks (line 240-248), thus validators would not go to next round (line 261-268). What even worse is `onCommitted` would reset the consensus timer in the end, so the malicious leader could use the same committed message to suspend the consensus process and eventually compromise the entire harmony network.

Also, consensus module uses `mapset.Set` to store received blocks / messages, but they are stored by their addresses, not contents (fields). So this vulnerability could also be exploited to attack committee members by flooding and make them Out-of-Memory.

Recommendation Add check when receiving peer's block height.

3.14 Missing Sanity Check on Slash Records - #1

- ID: PVE-014
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `staking/slash/double-sign.go`
- Category: Input Validation Issues [22]
- CWE subcategory: CWE-349 [23]

Description

This is a vulnerability in the slashing module, which could be exploited by attackers to compromise the harmony network consensus. Harmony network introduces Effective Proof-of-Stake, an efficient staking mechanism that avoids stake centralization while still supporting stake compounding and delegation.

In addition to the block rewards used to incentivize good behavior, the slashing mechanism is equally important as it can deter misbehavior and potential attacks. In EPoS, there are slashing rules for misbehaviors like double-signing or unavailability.

```

196 func (consensus *Consensus) onCommit(msg *msg_pb.Message) {
197     recvMsg, err := ParseFBFTMessage(msg)
198     log := consensus.getLogger()
199     if err != nil {
200         consensus.getLogger().Debug().Err(err).Msg("[OnCommit] Parse pbft message failed")
201         return
202     }
203
204     // NOTE let it handle its own log
205     if !consensus.isRightBlockNumAndViewID(recvMsg) {
206         return
207     }
208
209     consensus.mutex.Lock()
210     defer consensus.mutex.Unlock()
211
212     // TODO(audit): refactor into a new func
213     if key := (bls.PublicKey{}); consensus.couldThisBeADoubleSigner(recvMsg) {
214         if alreadyCastBallot := consensus.Decider.ReadBallot(
215             quorum.Commit, recvMsg.SenderPubkey,
216         ); alreadyCastBallot != nil {
217             for _, blk := range consensus.FBFTLog.GetBlocksByNumber(recvMsg.BlockNum) {
218                 alreadyCastBallot.SignerPubKey.ToLibBLSPublicKey(&key)
219                 if recvMsg.SenderPubkey.IsEqual(&key) {
220                     signed := blk.Header()
221                     areHeightsEqual := signed.Number().Uint64() == recvMsg.BlockNum
222                     areViewIDsEqual := signed.ViewID().Uint64() == recvMsg.ViewID
223                     areHeadersEqual := bytes.Compare(
224                         signed.Hash().Bytes(), recvMsg.BlockHash.Bytes(),
225                     ) == 0

```

Listing 3.41: consensus/leader.go

`onCommit` is called when leader receives commit messages from validators. It would perform lots of sanity checks to make sure the message is valid, also the submitter is not a double-signer (line 213-225).

```

332 func (w *Worker) CollectVerifiedSlashes() error {
333     pending, failures :=
334     w.chain.ReadPendingSlashingCandidates(), slash.Records{}

```

```

335     if d := pending; len(d) > 0 {
336         pending, failures = w.verifySlashes(d)
337     }
338
339     if f := failures; len(f) > 0 {
340         if err := w.chain.DeleteFromPendingSlashingCandidates(f); err != nil {
341             return err
342         }
343     }
344     w.current.slashes = pending
345     return nil
346 }

```

Listing 3.42: node/worker/worker.go

CollectVerifiedSlashes is responsible for collecting slashing evidences for double-signer which would be used later in block producing. However, the sanity check in CollectVerifiedSlashes could be bypassed and cause serious damages to the harmony network consensus.

```

153 func Verify(
154     chain CommitteeReader,
155     state *state.DB,
156     candidate *Record,
157 ) error {
158     wrapper, err := state.ValidatorWrapper(candidate.Offender)
159     if err != nil {
160         return err
161     }
162
163     if wrapper.EPOSStatus == effective.Banned {
164         return errAlreadyBannedValidator
165     }
166
167     if candidate.Offender == candidate.Reporter {
168         return errReporterAndOffenderSame
169     }
170
171     first, second :=
172         candidate.Evidence.AlreadyCastBallot,
173         candidate.Evidence.DoubleSignedBallot
174     k1, k2 := len(first.SignerPubKey), len(second.SignerPubKey)
175     if k1 != shard.PublicKeySizeInBytes ||
176        k2 != shard.PublicKeySizeInBytes {
177         return errors.Wrapf(
178             errSignerKeyNotRightSize, "cast key %d double-signed key %d", k1, k2,
179         )
180     }
181
182     if shard.CompareBlsPublicKey(first.SignerPubKey, second.SignerPubKey) != 0 {
183         k1, k2 := first.SignerPubKey.Hex(), second.SignerPubKey.Hex()
184         return errors.Wrapf(
185             errBallotSignerKeysNotSame, "%s %s", k1, k2,

```

```

186     )
187 }
188 currentEpoch := chain.CurrentBlock().Epoch()
189 // the slash can't come from the future (shard chain's epoch can't be larger than
    beacon chain's)
190 if candidate.Evidence.Epoch.Cmp(currentEpoch) == 1 {
191     return errors.Wrapf(
192         errSlashFromFutureEpoch, "current-epoch %v", currentEpoch,
193     )
194 }
195
196 superCommittee, err := chain.ReadShardState(candidate.Evidence.Epoch)
197
198 if err != nil {
199     return err
200 }
201
202 subCommittee, err := superCommittee.FindCommitteeByID(
203     candidate.Evidence.ShardID,
204 )
205
206 if err != nil {
207     return errors.Wrapf(
208         err, "given shardID %d", candidate.Evidence.ShardID,
209     )
210 }
211
212 if addr, err := subCommittee.AddressForBLSKey(
213     second.SignerPubKey,
214 ); err != nil || *addr != candidate.Offender {
215     return err
216 }
217
218 for _, ballot := range [...]votepower.Ballot{
219     candidate.Evidence.AlreadyCastBallot,
220     candidate.Evidence.DoubleSignedBallot,
221 } {
222     // now the only real assurance, cryptography
223     signature := &bls.Sign{}
224     publicKey := &bls.PublicKey{}
225
226     if err := signature.Deserialize(ballot.Signature); err != nil {
227         return err
228     }
229     if err := first.SignerPubKey.ToLibBLSPublicKey(publicKey); err != nil {
230         return err
231     }
232
233     blockNumBytes := make([]byte, 8)
234     // TODO(audit): add view ID into signature payload
235     binary.LittleEndian.PutUint64(blockNumBytes, ballot.Height)
236     commitPayload := append(blockNumBytes, ballot.BlockHeaderHash[:]...)

```



```

237     if !signature.VerifyHash(publicKey, commitPayload) {
238         return errFailVerifySlash
239     }
240 }
241
242 return nil
243 }

```

Listing 3.43: staking/slash/double-sign.go

However, the sanity check could be bypassed by providing two identical `votePower.ballot` since the loop (line 218-240) doesn't verify whether the two records are exactly the same. So an attacker could broadcast some crafted slash records, and the leader would take these records into account while producing new blocks, slash innocent validators and delegators, eventually compromise the whole harmony network.

Recommendation Add more checks when receiving slash records.

3.15 Missing Sanity Check on Slash Records - #2

- ID: PVE-015
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `core/blockchain.go`
- Category: Behavioral Problems [16]
- CWE subcategory: CWE-115 [21]

Description

This is a vulnerability in the slashing module, which could be exploited by attackers to compromise the harmony network consensus. Harmony introduces Effective Proof-of-Stake, an efficient staking mechanism that avoids stake centralization while still supporting stake compounding and delegation. In addition to the block rewards used to incentivize good behavior, the slashing mechanism is equally important as it can deter misbehavior and potential attacks. In EPoS, there are slashing rules for misbehavior like double-signing or unavailability.

```

196 func (consensus *Consensus) onCommit(msg *msg_pb.Message) {
197     recvMsg, err := ParseFBFTMessage(msg)
198     log := consensus.getLogger()
199     if err != nil {
200         consensus.getLogger().Debug().Err(err).Msg("[OnCommit] Parse pbft message failed")
201         return
202     }
203
204     // NOTE let it handle its own log
205     if !consensus.isRightBlockNumAndViewID(recvMsg) {

```

```

206     return
207 }
208
209 consensus.mutex.Lock()
210 defer consensus.mutex.Unlock()
211
212 // TODO(audit): refactor into a new func
213 if key := (bls.PublicKey{}); consensus.couldThisBeADoubleSigner(recvMsg) {
214     if alreadyCastBallot := consensus.Decider.ReadBallot(
215         quorum.Commit, recvMsg.SenderPubkey,
216     ); alreadyCastBallot != nil {
217         for _, blk := range consensus.FBFTLog.GetBlocksByNumber(recvMsg.BlockNum) {
218             alreadyCastBallot.SignerPubKey.ToLibBLSPublicKey(&key)
219             if recvMsg.SenderPubkey.IsEqual(&key) {
220                 signed := blk.Header()
221                 areHeightsEqual := signed.Number().Uint64() == recvMsg.BlockNum
222                 areViewIDsEqual := signed.ViewID().Uint64() == recvMsg.ViewID
223                 areHeadersEqual := bytes.Compare(
224                     signed.Hash().Bytes(), recvMsg.BlockHash.Bytes(),
225                 ) == 0

```

Listing 3.44: consensus/leader.go

`onCommit` is called when leader receives commit messages from validators. It would perform lots of sanity checks to make sure the message is valid, and the submitter is not a double-signer(line 213-225).

```

593 case doubleSign := <-node.Consensus.SlashChan:
594     utils.Logger().Info().
595         RawJSON("double-sign-candidate", []byte(doubleSign.String())).
596         Msg("double sign notified by consensus leader")
597     // no point to broadcast the slash if we aren't even in the right epoch yet
598     if !node.Blockchain().Config().IsStaking(
599         node.Blockchain().CurrentHeader().Epoch(),
600     ) {
601         return
602     }
603     if hooks := node.NodeConfig.WebHooks.Hooks; hooks != nil {
604         if s := hooks.Slashing; s != nil {
605             url := s.OnNoticeDoubleSign
606             go func() { webhooks.DoPost(url, &doubleSign) }()
607         }
608     }
609     if node.NodeConfig.ShardID != shard.BeaconChainShardID {
610         go node.BroadcastSlash(&doubleSign)
611     } else {
612         records := slash.Records{doubleSign}
613         if err := node.Blockchain().AddPendingSlashingCandidates(
614             records,
615         ); err != nil {
616             utils.Logger().Err(err).Msg("could not add new slash to ending slashes")
617         }

```

```

618 }
619 }

```

Listing 3.45: node/node.go

When double-signer is detected, beacon chain leader would call `AddPendingSlashingCandidates` to store the record (line 613); leaders of other shards would broadcast it to beacon chain through P2P message (line 610) .

```

39 func (bc *BlockChain) AddPendingSlashingCandidates(
40     candidates slash.Records,
41 ) error {
42     bc.pendingSlashingCandidatesMU.Lock()
43     defer bc.pendingSlashingCandidatesMU.Unlock()
44     current := bc.ReadPendingSlashingCandidates()
45     pendingSlashes := append(
46         bc.pendingSlashes, current.SetDifference(candidates)... ,
47     )
48     if l, c := len(pendingSlashes), len(current); l > maxPendingSlashes {
49         return errors.Wrapf(
50             errExceedMaxPendingSlashes, "current %d with-additional %d", c, l,
51         )
52     }
53     bc.pendingSlashes = pendingSlashes
54     return bc.writeSlashes(bc.pendingSlashes)
55 }

```

Listing 3.46: core/blockchain.go

`AddPendingSlashingCandidates` would make sure each slash record is unique (line 2038) and the length won't go beyond `maxPendingSlashes`. However, there is no sanity check to guarantee the slash records are valid, nor a limitation on how many records a node can send to others. So theoretically, a malicious node can flood a leader to stuff `bc.pendingSlashes` with lots of slash records, which could prevent legit slash records from being inserted into the slice and disable the slashing mechanism in a way. On the other hand, the uniqueness check can also be easily bypassed by adjusting some fields in the record, e.g., `TimeUnixNano`.

Recommendation Add more checks when adding slashing candidates.

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, after that, we examined the slash 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 confirmed and fixed, we do feel that the Harmony blockchain code has been thoroughly inspected, 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 Loop-hole. <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>.
- [18] MITRE. CWE-129: Improper Validation of Array Index. <https://cwe.mitre.org/data/definitions/129.html>.
- [19] MITRE. CWE-696: Incorrect Behavior Order. <https://cwe.mitre.org/data/definitions/696.html>.
- [20] MITRE. CWE CATEGORY: Integer Overflow or Wraparound. <https://cwe.mitre.org/data/definitions/190.html>.
- [21] MITRE. CWE-115: Misinterpretation of Input. <https://cwe.mitre.org/data/definitions/115.html>.
- [22] MITRE. CWE-1215: Input Validation Issues. <https://cwe.mitre.org/data/definitions/1215.html>.
- [23] MITRE. CWE-349: Acceptance of Extraneous Untrusted Data With Trusted Data. <https://cwe.mitre.org/data/definitions/349.html>.