SOFTWARE AUDIT REPORT

for

NERVOS FOUNDATION

Prepared By: Shuxiao Wang

Hangzhou, China

Nov. 8, 2019

Document Properties

Client	Nervos Foundation
Title	Software Audit Report
Target	Nervos CKB Blockchain
Version	1.0
Author	Jeff Liu
Auditors	Edward Lo, Huaguo Shi, Ruiyi Zhang, Wenbiao Zheng
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	Nov. 8, 2019	Jeff Liu	Final Release
1.0-rc2	Nov. 7, 2019	Jeff Liu	Release Candidate #2
1.0-rc1	Oct. 21, 2019	Jeff Liu	Release Candidate #1

Contact

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

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

Contents

1 Introduction

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

Nervos network [1] is a public blockchain system designed by Nervos Foundation [2]. Nervos is designed as a layered blockchain network, and it separates the network infrastructure into two layers: a verification layer (layer 1) as the consensus or common trust/knowledge storage layer, and a generation or computation layer (layer 2) for high throughput transaction generations. The goal of this layered design is to scale up the blockchain performance/throughput without sacrificing its security and decentralization.

Nervos CKB (Common Knowledge Base) [3] is the layer 1 blockchain, a public permission-less blockchain which generates trust and extends its trust to upper layer blockchains. CKB adopted a PoW-based optimized Nakamoto consensus (NC-Max) as its consensus algorithm to achieve maximized performance, and its virtual machine, CKB-VM, is compatible with RISC-V ISA. CKB testnet Rylai was launched on May 20th, 2019, and its mainnet is planned for launch by the end of the year.

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

Table 1.1: Basic Information of Nervos CKB Blockchain

Item	Description
Issuer	Nervos Foundation
Website	https://nervos.org
Type	Nervos CKB Blockchain
Platform	Rust
Audit Method	White-box
Latest Audit Report	Nov. 8, 2019

Table 1.2: The Commit Hash List Of Audited Branches

Git Repository	Commit Hash Of Audited Branch
https://github.com/nervosnetwork/ckb.git	253274db0c20e80294bd877d3aa94c3f33ac84d7

1.2 About PeckShield

PeckShield Inc. [4] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products including security audits. We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

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

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

Table 1.3: Vulnerability Severity Classification

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

- 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*, and *Low* shown in Table 1.3.

1.3.2 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by systematically finding and providing possible inputs to the target program, and then monitoring the program execution for crashes (or any unexpected results). In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing. As one of the most effective methods for exposing the presence of possible vulnerabilities, fuzzing technology has been the first choice for many security researchers in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to conduct fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the Nervos CKB Blockchain, we use AFL [6] 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 Nervos CKB Blockchain, we will further analyze it as part of the white-box audit.

1.3.3 White-box Audit

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

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

- Data and state storage, which is related to the database and files where blockchain data are saved.
- P2P networking, consensus, and transaction model in the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.
- VM, account model, and incentive model. This is essentially the execution and business layer of the blockchain, and many blockchain business specific logics are implemented here.
- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.

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
Incentive Model	Mining Algorithm Security
	Mining Algorithm ASIC Resistance
	Tokenization Reward Mechanism
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

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

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

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.5 to classify our findings.

1.4 Disclaimer

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

Table 1.5: Common Weakness Enumeration (CWE) Classifications Used in This Audit

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

2 Findings

2.1 Finding Summary

Here is a summary of our findings after analyzing Nervos CKB Blockchain. As mentioned earlier, we in the first phase of our audit studied CKB source code and ran our in-house static code analyzer through the codebase, and we focused on three CKB-VM implementations, namely `trace`, `ASM`, and `AOT`. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tools. After that, we manually review business logics, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

Besides the issues listed in the table, here we also include two screenshots of the current status of fuzzing. Figure 2.1 is a screenshot of a running AFL fuzzer. We examine these parameters regularly, and whenever the *uniq crashes* increases, we look into the input that triggers the new unique crash.

Table 2.1: The Severity of Our Findings

Severity	# of Findings	
Critical	4	■■■■
High	0	
Medium	5	■■■■■
Low	0	
Informational	3	■■■
Total	12	

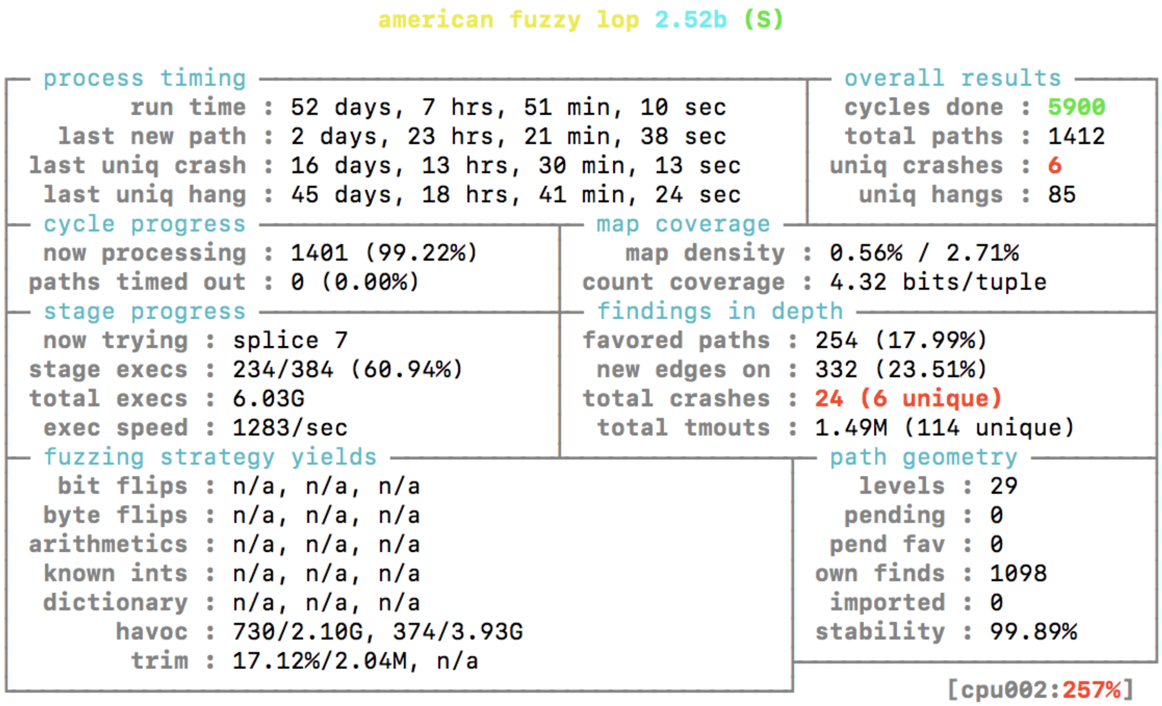
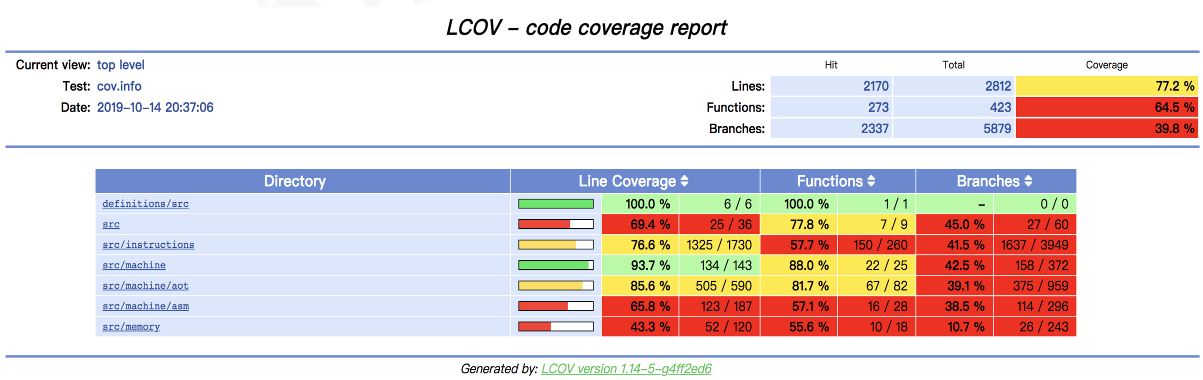


Figure 2.1: AFL Screenshot



LCOV – code coverage report

				Hit	Total	Coverage
Current view:	top level		Lines:	2170	2812	77.2 %
Test:	cov.info		Functions:	273	423	64.5 %
Date:	2019-10-14 20:37:06		Branches:	2337	5879	39.8 %

Directory	Line Coverage		Functions		Branches	
definitions/src	100.0 %	6 / 6	100.0 %	1 / 1	-	0 / 0
src	69.4 %	25 / 36	77.8 %	7 / 9	45.0 %	27 / 60
src/instructions	76.6 %	1325 / 1730	57.7 %	150 / 260	41.5 %	1637 / 3949
src/machine	93.7 %	134 / 143	88.0 %	22 / 25	42.5 %	158 / 372
src/machine/aot	85.6 %	505 / 590	81.7 %	67 / 82	39.1 %	375 / 959
src/machine/asm	65.8 %	123 / 187	57.1 %	16 / 28	38.5 %	114 / 296
src/memory	43.3 %	52 / 120	55.6 %	10 / 18	10.7 %	26 / 243

Generated by: LCOV version 1.14-5-g4ff2ed6

Figure 2.2: AFL Coverage

Once an issue that triggers a crash is determined to be valid, we follow up with additional investigation to identify possible root-causes and formulate fix recommendations for it.

At the same time, we also check the coverage metrics periodically and tune the **seed inputs** to cover more and more paths. Figure 2.2 illustrates the coverage results for the time being.

2.2 Key Findings

Table 2.2: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Misaligned Behavior #1 Between CKB-VM and RISC-V Specification	Coding Practices	Resolved
PVE-002	Medium	Misaligned Behavior #2 Between CKB-VM and RISC-V Specification	Coding Practices	Resolved
PVE-003	Medium	Misaligned Behavior #3 Between CKB-VM and RISC-V Specification	Coding Practices	Resolved
PVE-004	Medium	Integer Overflow in SupportMachine	Numeric Errors	Resolved
PVE-005	Critical	Integer Overflow in AsmCoreMachine	Numeric Errors	Resolved
PVE-006	Critical	DoS in LabelGatheringMachine	Error Conditions, Return Values, Status Codes	Resolved
PVE-007	Informational	Integer Overflow in LabelGatheringMachine	Numeric Errors	Resolved
PVE-008	Critical	Integer Overflow in the Flat Memory Module	Numeric Errors	Resolved
PVE-009	Critical	Integer Overflow in the load_data_as_code Syscall	Numeric Errors	Resolved
PVE-010	Medium	Reachable Assertion in the P2P Module	Error Conditions, Return Values, Status Codes	Resolved
PVE-011	Informational	Insufficient Risk Prompt in the CLI Module	Security Features	Resolved
PVE-012	Informational	Trade-Offs in the CKB Economic Model	Business Logics	Open

After analyzing all of the potential issues found during the audit, we determined that a number of them need to be brought up and paid more attention to, as shown in Table 2.2. Please refer to Section 3 for detailed discussion of each issue.

3 Detailed Results

3.1 Misaligned Behavior #1 Between CKB-VM and RISC-V Specification

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ckb-vm/src/instructions/rvc.rs`
- Category: Coding Practices [8]
- CWE subcategory: CWE-684 [9]

Description

CKB-VM is a pure software-only implementation of the RISC-V instruction set used for scripting VM in CKB. Right now it implements full IMC instructions for both 32-bit and 64-bit register size support. However, there is a discrepancy in the implementation of opcode `RVC_SRLI`.

According to the RISC-V Specification version 2.2 [10], opcode `RVC_SRLI` has the following format illustrated in Figure 3.1. Note that `RVC_SRLI` is an instruction that performs a logical right shift of the value in register rd' and then writes the result to rd'. The shift amount is encoded in the `shamt` field, where `shamt[5]` must be zero for the 32-bit architecture.

In current code base, the underlying logic of `RVC_SRLI` is implemented in `common::srli`.

```
    insts::OP_RVC_SRLI => {
        let i = Itype(inst);
        common::srli(machine, i.rd(), i.rs1(), i.immediate());
        None
    }
```

Listing 3.1: ckb-vm/src/instructions/execute.rs

```
pub fn srli<Mac: Machine>(
    machine: &mut Mac,
    rd: RegisterIndex,
    rs1: RegisterIndex,
    shamt: UImmediate,
```

```
) {
    let value = machine.registers()[rs1 as usize].clone() >> Mac::REG::from_u32(shamt);
    update_register(machine, rd, value);
}
```

Listing 3.2: ckb-vm/src/instructions/common.rs

The logic is rather straightforward: it basically performs right shift `i.rs1()` for `i.immediate()` bits. Note that `rs1()` and `immediate()` return corresponding fields of a opcode, and they are decoded as follows:

```
// [12]  => imm[5]
// [6:2] => imm[4:0]
fn uimmediate(instruction_bits: u32) -> u32 {
    (x(instruction_bits, 2, 5, 0) | x(instruction_bits, 12, 1, 5))
}
```

Listing 3.3: ckb-vm/src/instructions/rvc.rs

```
    let uimm = uimmediate(instruction_bits);
    match (instruction_bits & 0b_11_000_00000_00, uimm) {
    // Invalid instruction
    (0b_00_000_00000_00, 0) => None,
    // SRLI
    (0b_00_000_00000_00, uimm) => {
        Some(Itype::new(insts::OP_RVC_SRLI, rd, rd, uimm).0)
    }
    // Invalid instruction
    (0b_01_000_00000_00, 0) => None,
    // SRAI
    (0b_01_000_00000_00, uimm) => {
        Some(Itype::new(insts::OP_RVC_SRAI, rd, rd, uimm).0)
    }
    // ANDI
    (0b_10_000_00000_00, _) => Some(
        Itype::new_s(insts::OP_RVC_ANDI, rd, rd, immediate(instruction_bits)).0,
    ),
    _ => None,
    }
```

Listing 3.4: ckb-vm/src/instructions/rvc.rs

As mentioned in the RISC-V Specification version 2.2 [10], `shamt[5]` must be zero for the 32-bit architecture. However, the decoder does not have any related sanity check, hence resulting in

15–13	12	11–10	9–7	6–2	1–0
funct3	shamt[5]	funct2	rd′/rs1′	shamt[4:0]	op
3	1	2	3	5	2
C.SRLI	shamt[5]	C.SRLI	dest	shamt[4:0]	C1
C.SRAI	shamt[5]	C.SRAI	dest	shamt[4:0]	C1

Figure 3.1: Format of OPCode SRLI / SRAI

different behaviors between CKB-VM implementation and RISC-V specification.

Recommendation Add a sanity check or integer mask for the immediate part of the opcode.

```
    let uimm = uimmediate(instruction_bits);
    match (instruction_bits & 0b_11_000_00000_00, uimm) {
    // Invalid instruction
    (0b_00_000_00000_00, 0) => None,
    // SRLI
    (0b_00_000_00000_00, uimm) => Some(
        Itype::new(insts::OP_RVC_SRLI, rd, rd, uimm & u32::from(R::SHIFT_MASK))
        .0,
    ),
    // Invalid instruction
    (0b_01_000_00000_00, 0) => None,
    // SRAI
    (0b_01_000_00000_00, uimm) => Some(
        Itype::new(insts::OP_RVC_SRAI, rd, rd, uimm & u32::from(R::SHIFT_MASK))
        .0,
    ),
    // ANDI
    (0b_10_000_00000_00, _) => Some(
        Itype::new_s(insts::OP_RVC_ANDI, rd, rd, immediate(instruction_bits)).0,
    ),
    _ => None,
    }
```

Listing 3.5: ckb-vm/src/instructions/rvc.rs

3.2 Misaligned Behavior #2 Between CKB-VM and RISC-V Specification

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ckb-vm/src/instructions/rvc.rs`
- Category: Coding Practices [8]
- CWE subcategory: CWE-684 [9]

Description

CKB-VM is a pure software-only implementation of the RISC-V instruction set used for scripting VM in CKB. Right now it implements full IMC instructions for both 32-bit and 64-bit register size support. However, there is a discrepancy in the implementation of opcode `RVC_SRAI`.

According to the RISC-V Specification version 2.2 [10], opcode `RVC_SRAI` has the format illustrated in Figure 3.1. Specifically, `RVC_SRAI` is an instruction that performs a arithmetic right shift of the value in register rd' and then writes the result to rd'. The shift amount is encoded in the `shamt` field, where `shamt[5]` must be zero for the 32-bit architecture.

In the current code base, the underlying logic of `RVC_SRAI` is implemented in `common::srai`.

```
    insts::OP_RVC_SRAI => {
        let i = Itype(inst);
        common::srai(machine, i.rd(), i.rs1(), i.immediate());
        None
    }
```

Listing 3.6: ckb-vm/src/instructions/execute.rs

```
pub fn srai<Mac: Machine>(
    machine: &mut Mac,
    rd: RegisterIndex,
    rs1: RegisterIndex,
    shamt: UImmediate,
) {
    let value = machine.registers()[rs1 as usize].signed_shr(&Mac::REG::from_u32(shamt));
    update_register(machine, rd, value);
}
```

Listing 3.7: ckb-vm/src/instructions/common.rs

The logic is rather straightforward: it basically performs right shift `i.rs1()` for `i.immediate()` bits. Note that `rs1()` and `immediate()` return corresponding fields of a opcode, and they are decoded as follows:

```
// [12]  => imm[5]
// [6:2] => imm[4:0]
fn uimmediate(instruction_bits: u32) -> u32 {
    (x(instruction_bits, 2, 5, 0) | x(instruction_bits, 12, 1, 5))
}
```

Listing 3.8: ckb-vm/src/instructions/rvc.rs

```
    let uimm = uimmediate(instruction_bits);
    match (instruction_bits & 0b_11_000_00000_00, uimm) {
    // Invalid instruction
    (0b_00_000_00000_00, 0) => None,
    // SRLI
    (0b_00_000_00000_00, uimm) => {
        Some(Itype::new(insts::OP_RVC_SRLI, rd, rd, uimm).0)
    }
    // Invalid instruction
    (0b_01_000_00000_00, 0) => None,
    // SRAI
    (0b_01_000_00000_00, uimm) => {
        Some(Itype::new(insts::OP_RVC_SRAI, rd, rd, uimm).0)
    }
    // ANDI
```

```
    (0b_10_000_00000_00, _) => Some(
        Itype::new_s(insts::OP_RVC_ANDI, rd, rd, immediate(instruction_bits)).0,
    ),
    _ => None,
    }
```

Listing 3.9: ckb-vm/src/instructions/rvc.rs

As mentioned in the RISC-V Specification version 2.2 [10], `shamt[5]` must be zero for the 32-bit architecture. However, the decoder does not have any related sanity check, hence resulting in different behaviors between CKB-VM implementation and RISC-V specification.

Recommendation Add a sanity check or integer mask for the immediate part of the opcode.

```
    let uimm = uimmediate(instruction_bits);
    match (instruction_bits & 0b_11_000_00000_00, uimm) {
    // Invalid instruction
    (0b_00_000_00000_00, 0) => None,
    // SRLI
    (0b_00_000_00000_00, uimm) => Some(
        Itype::new(insts::OP_RVC_SRLI, rd, rd, uimm & u32::from(R::SHIFT_MASK))
        .0,
    ),
    // Invalid instruction
    (0b_01_000_00000_00, 0) => None,
    // SRAI
    (0b_01_000_00000_00, uimm) => Some(
        Itype::new(insts::OP_RVC_SRAI, rd, rd, uimm & u32::from(R::SHIFT_MASK))
        .0,
    ),
    // ANDI
    (0b_10_000_00000_00, _) => Some(
        Itype::new_s(insts::OP_RVC_ANDI, rd, rd, immediate(instruction_bits)).0,
    ),
    _ => None,
    }
```

Listing 3.10: ckb-vm/src/instructions/rvc.rs

3.3 Misaligned Behavior #3 Between CKB-VM and RISC-V Specification

- ID: PVE-003
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ckb-vm/src/instructions/rvc.rs`
- Category: Coding Practices [8]
- CWE subcategory: CWE-684 [9]

Description

CKB-VM is a pure software-only implementation of the RISC-V instruction set used for scripting VM in CKB. Right now it implements full IMC instructions for both 32-bit and 64-bit register size support. However, there is a discrepancy in the implementation of opcode `RVC_SLLI`.

According to the RISC-V Specification version 2.2 [10], opcode `RVC_SLLI` has the format illustrated in Figure 3.2. Specifically, `RVC_SLLI` is an instruction that performs a logical left shift of the value in register rd' and then writes the result to rd'. The shift amount is encoded in the `shamt` field, where `shamt[5]` must be zero for the 32-bit architecture.

In the current code base, the underlying logic of `RVC_SLLI` is implemented in `common::slli`:

```
    insts::OP_RVC_SLLI => {
        let i = Itype(inst);
        common::slli(machine, i.rd(), i.rs1(), i.immediate());
        None
    }
```

Listing 3.11: ckb-vm/src/instructions/execute.rs

```
pub fn slli<Mac: Machine>(
    machine: &mut Mac,
    rd: RegisterIndex,
    rs1: RegisterIndex,
    shamt: UImmediate,
) {
    let value = machine.registers()[rs1 as usize].clone() << Mac::REG::from_u32(shamt);
    update_register(machine, rd, value);
}
```

Listing 3.12: ckb-vm/src/instructions/common.rs

15 – 13	12	11 – 7	6 – 2	1 – 0
funct3	shamt[5]	rd/rs1	shamt[4:0]	op
3	1	5	5	2
C.SLLI	shamt[5]	dest≠0	shamt[4:0]	C2

Figure 3.2: Format of OPCode SLLI

The logic is rather straightforward: it basically performs left shift `i.rs1()` for `i.immediate()` bits. `rs1()` and `immediate()` return corresponding fields of a opcode, and it is decoded as follows:

```
// [12]   => imm[5]
// [6:2]  => imm[4:0]
fn uimmediate(instruction_bits: u32) -> u32 {
    (x(instruction_bits, 2, 5, 0) | x(instruction_bits, 12, 1, 5))
}
```

Listing 3.13: ckb-vm/src/instructions/rvc.rs

```
    let uimm = uimmediate(instruction_bits);
    let rd = rd(instruction_bits);
    if rd == 0 {
        // Reserved
        None
    } else if uimm != 0 {
        Some(Itype::new(insts::OP_RVC_SLLI, rd, rd, uimm).0)
    } else {
        Some(blank_instruction(insts::OP_RVC_SLLI64))
    }
```

Listing 3.14: ckb-vm/src/instructions/rvc.rs

As mentioned in the RISC-V Specification version 2.2 [10], `shamt[5]` must be zero for the 32-bit architecture. However, the decoder does not have any related sanity check, hence resulting in different behaviors between CKB-VM implementation and RISC-V specification.

Recommendation Add a sanity check or integer mask for the immediate part of the opcode.

```
    let uimm = uimmediate(instruction_bits);
    let rd = rd(instruction_bits);
    if rd == 0 {
        // Reserved
        None
    } else if uimm != 0 {
        Some(Itype::new(insts::OP_RVC_SLLI, rd, rd, uimm & u32::from(R::SHIFT_MASK)).0)
    } else {
        Some(blank_instruction(insts::OP_RVC_SLLI64))
    }
```

Listing 3.15: ckb-vm/src/instructions/rvc.rs

3.4 Integer Overflow in SupportMachine

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ckb-vm/src/machine/mod.rs`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [12]

Description

Given an ELF file, the CKB-VM uses the ELF loader module to parse the ELF header in the file. However, the ELF header, including its various member fields, should be validated. Unfortunately, the validation of certain member fields is not in place and these member fields, when being used in arithmetic calculation, may cause unexpected integer overflow and lead to VM crashes.

```
fn load_elf(&mut self, program: &Bytes, update_pc: bool) -> Result<(), Error> {
    let elf = Elf::parse(program).map_err(|_e| Error::ParseError)?;
    let bits = elf_bits(&elf.header).ok_or(Error::InvalidElfBits)?;
    if bits != Self::REG::BITS {
        return Err(Error::InvalidElfBits);
    }
    for program_header in &elf.program_headers {
        if program_header.p_type == PT_LOAD {
            let aligned_start = round_page_down(program_header.p_vaddr);
            let padding_start = program_header.p_vaddr - aligned_start;
            let size = round_page_up(program_header.p_memsz + padding_start);
            let slice_start = program_header.p_offset;
            let slice_end = program_header.p_offset + program_header.p_filesz;
            if slice_start > slice_end || slice_end > program.len() as u64 {
                return Err(Error::OutOfBound);
            }
            self.memory_mut().init_pages(
                aligned_start,
                size,
                convert_flags(program_header.p_flags)?,
                Some(program.slice(slice_start as usize, slice_end as usize)),
                padding_start,
            )?;
            self.memory_mut()
                .store_byte(aligned_start, padding_start, 0)?;
        }
    }
    if update_pc {
        self.set_pc(Self::REG::from_u64(elf.header.e_entry));
    }
    Ok(())
}
```

Listing 3.16: ckb-vm/src/machine/mod.rs

Specifically, within `load_elf()` (line 105), the loader module does not validate the header field values retrieved from the ELF file, which should not be trusted. Therefore, attackers might craft an ELF file with certain malicious header field values. For example, if an attacker sets `program_header.p_memsz` to `0xFFFFFFFFFFFFFFFF` (max unsigned value on 64-bit), the calculation will result in an integer overflow, which may cause a denial-of-service VM crash.

Meanwhile, it is important to note that when CKB-VM is compiled in the release mode, Rust does not include checks for integer overflows that cause panics. Instead, if an overflow occurs, Rust performs two's complement wrapping. As a result, CKB-VM may not panic, but the result is probably not expected. Fortunately, the consequence of this issue is not necessarily elusive (with *Low* impact), and we have so far not identified any interesting way to exploit it. For improved coding practices, developers are strongly encouraged to use the `wrapping_add()` function to avoid undesirable results, especially in security-sensitive scenarios.

Recommendation Add necessary sanity checks by wrapping the boundary of the result when using certain header field values.

```rust
105     let aligned_start = round_page_down(program_header.p_vaddr);
106     let padding_start = program_header.p_vaddr.wrapping_sub(aligned_start);
107     let size = round_page_up(program_header.p_memsz.wrapping_add(padding_start));
108     let slice_start = program_header.p_offset;
109     let slice_end = program_header
110         .p_offset
111         .wrapping_add(program_header.p_filesz);
112     if slice_start > slice_end || slice_end > program.len() as u64 {
113         return Err(Error::OutOfBound);
114     }
```

Listing 3.17: ckb-vm/src/machine/mod.rs

3.5 Integer Overflow in AsmCoreMachine

- ID: PVE-005
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `ckb-vm/src/machine/asm/mod.rs`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [12]

Description

Given an ELF file, the CKB-VM uses the ELF parser module to parse all sections in the file. However, the lack of necessary sanity checks for certain ELF header members, such as `header.e_entry` field, may be exploited to perform DoS attacks against CKB-VM.

```
122 fn execute_load16(&mut self, addr: u64) -> Result<u16, Error> {
123     check_permission(self, addr, 2, FLAG_EXECUTABLE)?;
124     self.load16(&(addr)).map(|v| v as u16)
125 }
```

Listing 3.18: ckb-vm/src/machine/asm/mod.rs

Specifically, within `execute_load16()` (line 123), it calls `check_permission` to ensure the memory page is either *writable* or *executable*.

```
pub fn check_permission<R: Register>(
    memory: &mut Memory<R>,
    addr: u64,
    size: u64,
    flag: u8,
  ) -> Result<(), Error> {
    let e = addr + size;
    let mut current_addr = round_page_down(addr);
    while current_addr < e {
        let page = current_addr / RISCV_PAGESIZE as u64;
        let page_flag = memory.fetch_flag(page)?;
        if (page_flag & FLAG_WXORX_BIT) != (flag & FLAG_WXORX_BIT) {
            return Err(Error::InvalidPermission);
        }
        current_addr += RISCV_PAGESIZE as u64;
    }
    Ok(())
}
```

Listing 3.19: ckb-vm/src/memory/mod.rs

As shown in the code snippet above, `check_permission` basically checks the permission of each page starting from `addr` to `addr + 2` (in the case of `execute_load16()`). However, attackers might craft an ELF with a malicious entry point address. In particular, if an attacker sets the `addr` to `0xFFFFFFFFFFFFFFFF` (max unsigned value on 64-bit), the calculation will result in an addition overflow and crash VM execution when calling `load16` (line 140).

```
135  fn load16(&mut self, addr: &u64) -> Result<u64, Error> {
136     let addr = *addr;
137     if addr + 2 > self.memory.len() as u64 {
138         return Err(Error::OutOfBound);
139     }
140     Ok(u64::from(LittleEndian::read_u16(
141         &self.memory[addr as usize..addr as usize + 2],
142     )))
143  }
```

Listing 3.20: ckb-vm/src/machine/asm/mod.rs

Recommendation Add necessary sanity checks to validate the requested memory is within proper bounds.

```rust
pub fn check_permission<R: Register>(
    memory: &mut dyn Memory<R>,
    addr: u64,
    size: u64,
    flag: u8,
) -> Result<(), Error> {
    // fetch_flag below will check if requested memory is within bound. Here
    // we only need to test for overflow first
    let (e, overflowed) = addr.overflowing_add(size);
    if overflowed {
        return Err(Error::OutOfBound);
    }
    let mut current_addr = round_page_down(addr);
    while current_addr < e {
        let page = current_addr / RISCV_PAGESIZE as u64;
        let page_flag = memory.fetch_flag(page)?;
        if (page_flag & FLAG_WXORX_BIT) != (flag & FLAG_WXORX_BIT) {
            return Err(Error::InvalidPermission);
        }
        current_addr += RISCV_PAGESIZE as u64;
    }
    Ok(())
}
```

Listing 3.21: ckb-vm/src/memory/mod.rs

3.6 Denial-of-Service in LabelGatheringMachine

- ID: PVE-006
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `ckb-vm/src/machine/aot/mod.rs`
- Category: Error Conditions, Return Values, Status Codes [13]
- CWE subcategory: CWE-248 [14]

Description

`LabelGatheringMachine` is an AOT [15] implementation of CKB-VM, which compiles higher-level code into machine-optimized code before execution for improved runtime performance. Particularly, when given an ELF file, the `LabelGatheringMachine` VM uses the `load` function to parse all sections in the file. However, a malformed ELF file may fail the parser and further crash CKB-VM execution.

```rust
pub fn load(program: &Bytes) -> Result<Self, Error> {
    let elf = Elf::parse(&program).unwrap();
    if elf.section_headers.len() > MAXIMUM_SECTIONS {
        return Err(Error::LimitReached);
    }
    ...
```

```
101 }
```

Listing 3.22: ckb-vm/src/machine/aot/mod.rs

Specifically, an attacker may craft an invalid or malformed ELF file that causes the `Elf::parse()` in line 96 to return an `Error`. Unfortunately, in current implementation of Rust's `unwrap()`, the process simply panics if the value is an `Error`:

```
impl<T, E: fmt::Debug> Result<T, E> {
    ...
    #[inline]
    #[stable(feature = "rust1", since = "1.0.0")]
    pub fn unwrap(self) -> T {
        match self {
            Ok(t) => t,
            Err(e) => unwrap_failed("called `Result::unwrap()` on an `Err` value", &e),
        }
    }
}
```

Listing 3.23: rust/src/libcore/result.rs firstnumber

Recommendation Add necessary sanity checks to properly handle the return value of `Elf::parse()`.

```
95  pub fn load(program: &Bytes) -> Result<Self, Error> {
96      let elf = Elf::parse(program).map_err(|_e| Error::ParseError)?;
97      if elf.section_headers.len() > MAXIMUM_SECTIONS {
98          return Err(Error::LimitReached);
99      }
100     ...
101 }
```

Listing 3.24: ckb-vm/src/machine/aot/mod.rs

3.7 Integer Overflow in LabelGatheringMachine

- ID: PVE-007
- Severity: Informational
- Likelihood: High
- Impact: None
- Target: `ckb-vm/src/machine/aot/mod.rs`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [12]

Description

As mentioned earlier, `LabelGatheringMachine` is an AOT [15] implementation of CKB-VM, which compiles higher-level code into machine-optimized code before execution for improved runtime performance. Particularly, when given an ELF file, the `LabelGatheringMachine` VM uses the `load` function to parse all sections based on the `section_header` fields contained in the file. However, these `section_header` member fields may not be trustworthy and their uses must be validated. Unfortunately, such validation for certain member fields is not in place and these member fields, when being used in arithmetic calculation, may cause unexpected integer overflow and lead to VM crashes.

```
95  pub fn load(program: &Bytes) -> Result<Self, Error> {
96      let elf = Elf::parse(&program).unwrap();
97      if elf.section_headers.len() > MAXIMUM_SECTIONS {
98          return Err(Error::LimitReached);
99      }
100     let mut sections: Vec<(u64, u64)> = elf
101         .section_headers
102         .iter()
103         .filter_map(|section_header| {
104             if section_header.sh_flags & u64::from(SHF_EXECINSTR) != 0 {
105                 Some((
106                     section_header.sh_addr,
107                     section_header.sh_addr + section_header.sh_size,
108                 ))
109             } else {
110                 None
111             }
112         })
113         .rev()
114         .collect();
115     // Test there's no empty section
116     if sections.iter().any(|(s, e)| s >= e) {
117         return Err(Error::OutOfBound);
118     }
119     ...
120 }
```

Listing 3.25: ckb-vm/src/machine/aot/mod.rs

Specifically, the `start`/`end` addresses of each executable section are collected in the iteration (lines 100-114). However, an attacker may craft an ELF with certain malicious section header values. In particular, if she initializes `section_header.sh_addr` to `0xFFFFFFFFFFFFFFFF` (max unsigned value on 64-bit), the calculation will result in an addition overflow, which may cause a denial-of-service VM crash.

Meanwhile, it is important to note that when CKB-VM is compiled in the release mode, Rust does not include checks for integer overflows that cause panics. Instead, if an overflow occurs, Rust performs two's complement wrapping. As a result, CKB-VM may not panic, but the result is probably not expected. Fortunately, there is an additional sanity check within the same function that detects and blocks such overflow (line 116 in the above code snippet). For this very reason, the impact of this issue is considered *None* and the overall severity is reduced to *Informational*. But for improved coding practices, developers are still strongly encouraged to use the `wrapping_add()` function to avoid undesirable results, especially in security-sensitive scenarios.

Recommendation Add necessary sanity checks by wrapping the boundary of the result of `section_header.sh_addr + section_header.sh_size`.

```
pub fn load(program: &Bytes) -> Result<Self, Error> {
    let elf = Elf::parse(&program).unwrap();
    if elf.section_headers.len() > MAXIMUM_SECTIONS {
        return Err(Error::LimitReached);
    }
    let mut sections: Vec<(u64, u64)> = elf
        .section_headers
        .iter()
        .filter_map(|section_header| {
            if section_header.sh_flags & u64::from(SHF_EXECINSTR) != 0 {
                Some((
                    section_header.sh_addr,
                    section_header.sh_addr.wrapping_add(section_header.sh_size),
                ))
            } else {
                None
            }
        })
        .rev()
        .collect();
    ...
    // Test there's no empty section
    if sections.iter().any(|(s, e)| s >= e) {
        return Err(Error::OutOfBound);
    }
}
```

Listing 3.26: ckb-vm/src/machine/aot/mod.rs

3.8 Integer Overflow in the Flat Memory Module

- ID: PVE-008
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `ckb-vm/src/memory/flat.rs`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [12]

Description

CKB-VM is a pure software-only implementation of the RISC-V instruction set used for scripting VM in CKB. It supports the so-called flat memory model that simply uses a flat chunk of memory used for RISC-V machine and thus lacks all the permission checking logic. In this flat memory model, when an ELF file is being loaded, CKB-VM parses all program segments based on the `program_header` fields contained in the file. However, these `program_header` member fields may not be trustworthy and their uses must be validated. Unfortunately, such validation for certain member fields is not in place and these member fields, when being used in arithmetic calculation, may cause unexpected integer overflow and lead to VM crashes.

```
101 for program_header in &elf.program_headers {
102     if program_header.p_type == PT_LOAD {
103         let aligned_start = round_page_down(program_header.p_vaddr);
104         let padding_start = program_header.p_vaddr.wrapping_sub(aligned_start);
105         let size = round_page_up(program_header.p_memsz.wrapping_add(padding_start));
106         let slice_start = program_header.p_offset;
107         let slice_end = program_header
108             .p_offset
109             .wrapping_add(program_header.p_filesz);
110         if slice_start > slice_end || slice_end > program.len() as u64 {
111             return Err(Error::OutOfBound);
112         }
113         self.memory_mut().init_pages(
114             aligned_start,
115             size,
116             convert_flags(program_header.p_flags)?,
117             Some(program.slice(slice_start as usize, slice_end as usize)),
118             padding_start,
119         )?;
120         self.memory_mut()
121             .store_byte(aligned_start, padding_start, 0)?;
122     }
123 }
```

Listing 3.27: ckb-vm/src/machine/mod.rs

Specifically, for each loadable program segment (line 102), CKB-VM initializes its own memory segment. Note that he segment initialization routine differs for different memory models. In the case of the flat memory model, it directly calls `fill_page_data()`:

```
fn init_pages(
    &mut self,
    addr: u64,
    size: u64,
    _flags: u8,
    source: Option<Bytes>,
    offset_from_addr: u64,
) -> Result<(), Error> {
    fill_page_data(self, addr, size, source, offset_from_addr)
}
```

Listing 3.28: ckb-vm/src/memory/flat.rs

```
pub(crate) fn fill_page_data<R: Register>(
    memory: &mut dyn Memory<R>,
```

```
    addr: u64,
    size: u64,
    source: Option<Bytes>,
    offset_from_addr: u64,
) -> Result<(), Error> {
    let mut written_size = 0;
    if offset_from_addr > 0 {
        let real_size = min(size, offset_from_addr);
        memory.store_byte(addr, real_size, 0)?;
        written_size += real_size;
    }
    if let Some(source) = source {
        let real_size = min(size - written_size, source.len() as u64);
        if real_size > 0 {
            memory.store_bytes(addr + written_size, &source[0..real_size as usize])?;
            written_size += real_size;
        }
    }
    if written_size < size {
        memory.store_byte(addr + written_size, size - written_size, 0)?;
    }
    Ok(())
}
```

Listing 3.29: ckb-vm/src/memory/mod.rs

The `fill_page_data()` routine uses two other subroutines `store_byte()`/`store_bytes()` to load actual contents.

```
fn store_bytes(&mut self, addr: u64, value: &[u8]) -> Result<(), Error> {
    let size = value.len() as u64;
    if addr + size > self.len() as u64 {
        return Err(Error::OutOfBound);
    }
    let slice = &mut self[addr as usize..(addr + size) as usize];
    slice.copy_from_slice(value);
    Ok(())
}

fn store_byte(&mut self, addr: u64, size: u64, value: u8) -> Result<(), Error> {
    if addr + size > self.len() as u64 {
        return Err(Error::OutOfBound);
    }
    memset(&mut self[addr as usize..(addr + size) as usize], value);
    Ok(())
}
```

Listing 3.30: ckb-vm/src/memory/flat.rs

```
fn init_pages(
    &mut self,
    addr: u64,
    size: u64,
```

```
    flags: u8,
    source: Option<Bytes>,
    offset_from_addr: u64,
) -> Result<(), Error> {
    if round_page_down(addr) != addr || round_page_up(size) != size {
        return Err(Error::Unaligned);
    }
    if addr > RISCV_MAX_MEMORY as u64
        || size > RISCV_MAX_MEMORY as u64
        || addr + size > RISCV_MAX_MEMORY as u64
        || offset_from_addr > size
    {
        return Err(Error::OutOfBound);
    }
    ...
}
```

Listing 3.31: ckb-vm/src/memory/wxorx.rs

In the current code base, it lacks some essential sanity checks. Specifically, an attacker might craft a malformed ELF file with certain program header fields to trigger an addition overflow (lines 157 and 166) or a panic (lines 160 and 169).

Recommendation Add necessary sanity checks in the flat memory module's subroutines `store_byte()`/`store_bytes()`.

```
155 fn store_bytes(&mut self, addr: u64, value: &[u8]) -> Result<(), Error> {
156     let size = value.len() as u64;
157     if addr.checked_add(size).ok_or(Error::OutOfBound)? > self.len() as u64 {
158         return Err(Error::OutOfBound);
159     }
160     let slice = &mut self[addr as usize..(addr + size) as usize];
161     slice.copy_from_slice(value);
162     Ok(())
163 }
164
165 fn store_byte(&mut self, addr: u64, size: u64, value: u8) -> Result<(), Error> {
166     if addr.checked_add(size).ok_or(Error::OutOfBound)? > self.len() as u64 {
167         return Err(Error::OutOfBound);
168     }
169     memset(&mut self[addr as usize..(addr + size) as usize], value);
170     Ok(())
171 }
```

Listing 3.32: ckb-vm/src/memory/flat.rs

3.9 Integer Overflow in the load_data_as_code Syscall

- ID: PVE-009
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `ckb/script/src/syscalls/load_cell_data.rs`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [12]

Description

There is a vulnerability in the CKB syscall `load_data_as_code`, which could be exploited by attackers to perform a DoS attack and crash the CKB execution.

```
int ckb_load_cell_code(void* addr, size_t memory_size, size_t content_offset,
                       size_t content_size, size_t index, size_t source) {
    return syscall(SYS_ckb_load_cell_data_as_code, addr, memory_size,
                   content_offset, content_size, index, source);
}
```

Listing 3.33: ckb-system-scripts/c/ckb_syscalls.h

Specifically, when analyzing the above code snippet, we notice that CKB allows a contract to make syscalls by passing parameters of any value in the range of `size_t`. Different syscalls have different semantics when interpreting and handling their parameters. In the case of `load_data_as_code` syscall, it ensures that `content_offset + content_size` is smaller than `cell.data_bytes` (line 92):

```
92      if content_offset >= cell.data_bytes
93          || (content_offset + content_size) > cell.data_bytes
94          || content_size > memory_size
95      {
96          machine.set_register(A0, Mac::REG::from_u8(SLICE_OUT_OF_BOUND));
97          return Ok(());
98      }
99      let data = self
100         .data_loader
101         .load_cell_data(cell)
102         .ok_or(VMError::Unexpected)?
103         .0;
104     machine.memory_mut().init_pages(
105         addr,
106         memory_size,
107         FLAG_EXECUTABLE | FLAG_FREEZED,
108         Some(data.slice(
109             content_offset as usize,
110             (content_offset + content_size) as usize,
111         )),
112         0,
113     )?;
```

Listing 3.34: ckb/script/src/syscalls/load_cell_data.rs

However, these parameters are directly passed from a user-controlled transaction and thus they should be validated before their uses. Unfortunately, in current implementation, such validation is insufficient and malicious parameters, i.e., `content_offset` and `content_size`, can cause an addition overflow (lines 93 and 110).

Recommendation Add necessary sanity checks in the `load_data_as_code` syscall handler.

```
92      if content_offset >= cell.data_bytes
93          || (content_offset.saturating_add(content_size)) > cell.data_bytes
94          || content_size > memory_size
95      {
96          machine.set_register(A0, Mac::REG::from_u8(SLICE_OUT_OF_BOUND));
97          return Ok(());
98      }
99      let data = self
100         .data_loader
101         .load_cell_data(cell)
102         .ok_or(VMError::Unexpected)?
103         .0;
104     machine.memory_mut().init_pages(
105         addr,
106         memory_size,
107         FLAG_EXECUTABLE | FLAG_FREEZED,
108         Some(data.slice(
109             content_offset as usize,
110             (content_offset.saturating_add(content_size)) as usize,
111         )),
112         0,
113     )?;
```

Listing 3.35: ckb/script/src/syscalls/load_cell_data.rs

3.10 Reachable Assertion in the P2P Module

- ID: PVE-010
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `ckb-p2p/src/session.rs`
- Category: Error Conditions, Return Values, Status Codes [13]
- CWE subcategory: CWE-617 [16]

Description

`tentacle` is a multiplexed p2p network framework that implements on top of `yamux` to support mounting custom protocols and is considered the bedrock of the CKB P2P network. However, there is a reachable assertion in `tentacle`, which could be exploited by attackers to possibly cause denial-of-service attacks.

Specifically, once a client-server session is being established, the client is requested to negotiate protocol supporting information with the remote server with the goal of reaching a consensus for supported protocols and versions.

```
pub fn open_proto_stream(&mut self, proto_name: &str) {
    ...
    let task = client_select(handle, proto_info);
    self.select_procedure(task);
}
...
fn handle_sub_stream(&mut self, sub_stream: StreamHandle) {
    ...
    let task = server_select(sub_stream, proto_metas);
    self.select_procedure(task);
}
```

Listing 3.36: ckb-p2p/src/session.rs

By exchanging each supported protocol information, both the client and the server aim to simultaneously reach a consensus. When a consensus is reached, a protocol stream can then be opened to handle their specific communication.

```
fn select_procedure(
    &mut self,
    procedure: impl Future<
        Item = (
                Framed<StreamHandle, LengthDelimitedCodec>,
                String,
                Option<String>,
        ),
        Error = io::Error,
    > + Send
    + 'static,
) {
...
    match result {
        Ok((handle, name, version)) => match version {
            Some(version) => {
                let send_task = event_sender.send(ProtocolEvent::Open {
                    sub_stream: Box::new(handle),
                    proto_name: name,
                    version,
                });
                tokio::spawn(send_task.map(|_| ()).map_err(|err| {
                    debug!("stream send back error: {:?}", err);
                }));
            }
...
        },
...
fn handle_stream_event(&mut self, event: ProtocolEvent) {
    match event {
```

```
31         ProtocolEvent::Open {
32             proto_name,
33             sub_stream,
34             version,
35         } => {
36             self.open_protocol(proto_name, version, sub_stream);
37         }
38 ...
39 fn open_protocol(
40     &mut self,
41     name: String,
42     version: String,
43     sub_stream: Box<Framed<StreamHandle, LengthDelimitedCodec>>,
44 ) {
45     let proto = match self.protocol_configs.get(&name) {
46         Some(proto) => proto,
47         None => unreachable!(),
48     };
```

Listing 3.37: ckb-p2p/src/session.rs

However, there is a reachable assertion! If the server intentionally returns malicious protocol data with an arbitrary protocol name, the client side may blindly trust it. As indicated in line 47 of the above code snippet, it could lead to `unreachable!()` macro invocation and thus immediately crash its underlying `tokio` worker thread.

Recommendation Add a sanity check for the protocol name returned from the server side.

```
    let task = procedure.timeout(self.timeout).then(|result| {
        match result {
            Ok((handle, name, version)) => match version {
              Some(version) => {
              if let Some(proto) = self.protocol_configs.get(&name) {
                  ...
              } else {
                  ...
              }
            }
```

Listing 3.38: ckb-p2p/src/session.rs

3.11 Insufficient Risk Prompt in the CLI Module

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ckb-cli/src/subcommands/account.rs`
- Category: Security Features [17]
- CWE subcategory: CWE-260 [18]

Description

The CKB-CLI implements various wallet features and exports a number of blockchain API interfaces. However, there exists a potential risk to the user with using the subcommand `export` to dump in plaintext an account's private key.

```
>> account: Manage accounts

 list       List all accounts
 new        Create a new account and print related information.
 import     Import an unencrypted private key from <privkey-path> and create a new
     account.
 unlock     Unlock an account
 update     Update password of an account
 export     Export master private key and chain code as hex plain text (USE WITH YOUR
    OWN RISK)
```

Listing 3.39: ckb-cli/README.md

```
("export", Some(m)) => {
    let lock_arg: H160 =
        FixedHashParser::<H160>::default().from_matches(m, "lock-arg")?;
    let key_path = m.value_of("extended-privkey-path").unwrap();
    let password = read_password(false, None)?;

    if Path::new(key_path).exists() {
        return Err(format!("File exists: {}", key_path));
    }
    let master_privkey = self
        .key_store
        .export_key(&lock_arg, password.as_bytes())
        .map_err(|err| err.to_string())?;
    let bytes = master_privkey.to_bytes();
    let privkey = H256::from_slice(&bytes[0..32]).unwrap();
    let chain_code = H256::from_slice(&bytes[32..64]).unwrap();
    let mut file = fs::File::create(key_path).map_err(|err| err.to_string())?;
    file.write(format!("{:x}\n", privkey).as_bytes())
        .map_err(|err| err.to_string())?;
    file.write(format!("{:x}", chain_code).as_bytes())
        .map_err(|err| err.to_string())?;
   Ok(format!(
```

```
            "Success exported account as extended privkey to: \"{}\", please use this
                file carefully",
            key_path
        ))
    }
```

Listing 3.40: ckb-cli/src/subcommands/account.rs

As indicated in the above code snippet, CKB-CLI allows an account's private key to be exported and stored in clear text. Although the file `READ.md` indeed informs the user that storing the private key in plaintext format is risky, current prompt of `"Please use this file with caution"` can be strengthened to better raise potential risks of storing private keys in plaintext.

Recommendation It is strongly recommended to add a clear security risk warning. When using the `export` subcommand, the risk of exporting the private key is explicitly indicated in the output information, or it should send clear messages to users to avoid or even deprecate such usage.

3.12 CKB Economic Model Discussion: Mining Reward And ASIC Resistance

- ID: PVE-012
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: N/A
- Category: Business Logics [19]
- CWE subcategory: CWE-666 [20]

Description

A public blockchain is an open system with diverse participants and use scenarios, and the goal of its economic model is to provide incentives to all participants and align their own interests with the success of the blockchain. For PoW-based blockchains such as Bitcoin or CKB, The main participants include miners who contribute resource and maintain the network, and two main user groups that use the blockchain as a network for Medium of Exchange (MoE) or Store of Value (SoV). The well-being of the blockchain mainly benefits these user groups, therefore the economic model should provide a fair way for them to compensate the miners to cover their operating cost and for them to earn a reasonable profit.

The operators of the blockchain, miners, have to pay the enormous cost of operating the network to provide security and decentralization to the users of the blockchain. For the MoE users, they are only exposed to the security risk momentarily, so they may not be willing to pay for it; On the other hand, security and decentralization are highly valuable properties for the SoV users since their assets are stored on the network for prolonged period, therefore they may be more willing to pay for it. In the case of Bitcoin, MoE users pay a small transaction fee, but the main income of miners come from block reward, which essentially is a rent paid by SoV users in the form of a Bitcoin inflation tax.

An issue with Bitcoin mining reward/coin issuance scheme is that the block reward halves about every four years, as time goes by, the SoV users contribute less and less to cover the network operating cost although they are the main beneficiary, and until one day in the year of 2140, SoV users will not pay block reward anymore and all the cost will be covered by transaction fees paid by MoE users.

To improve from Bitcoin's scheme and achieve better economical fairness, on top of the transaction fee and block reward (base issuance), CKB added a secondary issuance. Unlike the base issuance, which also halves every four years like Bitcoin scheme and goes to miners entirely, the secondary issuance is a new coin issuance in a constant rate, and only a portion of it goes to miners depending on the CKB network usage at that time. Parts of the secondary issuance may go to coin holders or be burned so the coin holders not using the network would not pay the inflation tax.

Overall, CKB's coin issuance scheme is an improvement comparing to Bitcoin's, since it ensures SoV users to pay rent continuously even as the base issuance dwindles over time. Although in principle, both base and second issuances are rent paid by SoV users, and they are redundant and the sum of them may still go down over time, which is not aligned with the reality that the operating cost of a blockchain tends to go up year over year.

The CKB base issuance is 4.2 billion coins per year initially, and the second issuance is 1.344 billion coins per year according to the current CKB design. The intention is to let the base issuance being miner's main income in the early years of the CKB operation, and the second issuance takes over later as the base issuance decreases and the network utility rate goes up. We agree with the intention, although depends on the actual network utility rate over time, the secondary issuance may be too little or too much, and the optimal rate of the second issuance cannot be known before hand.

Another related economic issue is the ASIC-resistance of the mining algorithm because it affects the rate of decentralization of a blockchain and miners' willingness to participate. ASIC-resistance is a property of measuring how much a cryptocurency is "resistant" or "immune" to ASIC mining. ASIC-resistant cryptocurrency is that their mining algorithm is configured so that using ASIC machines to mine them does not bring significant benefit comparing with traditional GPU mining, or it's impossible to mine it using ASIC. In this sense, Bitcion is not an ASIC-resistant cryptocrrency. Instead of the `SHA256` hashing algorithm used by Bitcoin, CKB developed its own hashing algorithm, called `Eaglesong`. `Eaglesong` is a PoW hash function built with sponge construction similar to `SHA3`, and it is simple and requires low hardware investment. Comparing to Bitcoin, we feel that CKB's `Eaglesong` algorithm is more memory-bound instead of CPU-bound, which diminishes the ASIC advantage, and its simpler design also makes building ASIC easier so more people could do it in case some parties would like to try. Considering these reasons, we believe `Eaglesong` is ASIC-neutral or somewhat ASIC-resistant.

4 | Conclusion

In this security audit, we have analyzed the Nervos CKB 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 P2P networking, consensus algorithm, transaction model, and economic model, etc. 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 12 issues need to be brought up and paid more attention to, which are reported in Sections 2 and 3.

Our impression through this audit is that the Nervos CKB Blockchain software is neatly organized and elegantly implemented and those identified issues are promptly confirmed and fixed. We'd like to commend Nervos Foundation 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] Nervos Foundation. Nervos Network. https://github.com/nervosnetwork/.

[2] Nervos Foundation. Nervos Foundation. https://nervos.org.

[3] Nervos Foundation. Nervos CKB. https://github.com/nervosnetwork/ckb.

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

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

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

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

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

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

[10] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual Volume I: User-Level ISA. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

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

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

[13] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[14] MITRE. CWE-248: Uncaught Exception. https://cwe.mitre.org/data/definitions/248.html.

[15] Wikipedia. Ahead of time compilation. https://en.wikipedia.org/wiki/Ahead-of-time_compilation.

[16] MITRE. CWE-617: Reachable Assertion. https://cwe.mitre.org/data/definitions/617.html.

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

[18] MITRE. CWE-260: Password in Configuration File. https://cwe.mitre.org/data/definitions/260.html.

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

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