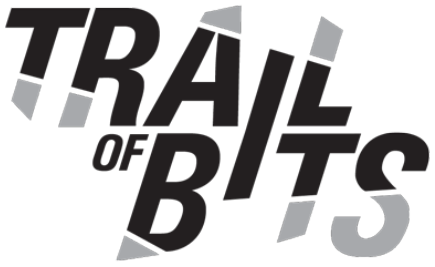


RSKj

Security Assessment

November 13, 2017

Prepared For:
RSK Labs

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Evan Sultanik | *Trail of Bits*
evan.sultanik@trailofbits.com

Changelog

November 13, 2017:	Initial report delivered
November 30, 2017:	Added Appendix F with retest results
December 4, 2017:	Updated Appendix F with notes about EIP 161

Executive Summary

RSK engaged Trail of Bits to perform an audit of RSKj, its solution for providing smart contracts using the Bitcoin cryptocurrency. Two Trail of Bits researchers performed the assessment from October 23rd through November 10th, 2017. The engagement was conducted as a source code review focusing on smart contract issues related to the virtual machine, virtual machine compatibility with other Ethereum implementations, correctness of the Trie data structure, and correctness of the precompiled contracts for the two-way peg. RSK also expressed concern about general susceptibility to denial of service attacks.

Trail of Bits completed the assessment using static and dynamic analysis techniques over a period of three calendar weeks. During the first week, Trail of Bits gained an understanding of the VM architecture and how to interact with and test its implementation. During the second week, Trail of Bits reviewed the correctness of the EVM opcode implementations, their gas consumption, and the Trie data structure implementation. For the last week, Trail of Bits continued the analysis of EVM opcode implementations, attempted to break the virtual machine through exceptions, and audited the precompiled smart contracts, including the two-way peg bridge.

The assessment identified a variety of issues in RSKj, including two high-severity findings in the EVM implementation. These issues would allow an attacker to craft smart contracts that are secure in Ethereum, but contain a backdoor when run in RSKj. As a result, an attacker could fool users and steal ether. As these issues are all related to the EVM implementation, we expect that similar issues may be present. Also, we report less-severe vulnerabilities related to the EVM implementation and possible denial of service due to memory leaks caused by improper resource management.

We discovered several concurrency issues where guarded variables are manipulated without proper lock synchronization. Furthermore, the code contains several common Java errors such as using primitive arrays as map keys and classes that override `equals` but not `hashCode`. These code quality issues and others like them may lead to the introduction of future vulnerabilities, discussed in Appendix C. Therefore, Trail of Bits recommends that RSK consider these issues and enforce coding standards to eliminate them.

The discovered vulnerabilities and the current state of the platform are as expected. RSKj is a complex system that integrates and extends several disparate technologies. Like any complex system, a multitude of corner cases must be handled properly in order to avoid potential bugs. An effort has been made to address specific areas of concern, such as integer overflows in gas computation. Further efforts are required to properly protect RSKj against other attack vectors. RSKj should correct the identified vulnerabilities, follow strict coding rules, and anticipate future vulnerability discovery by adding thorough unit tests.

Project Dashboard

Application Summary

Name	RSKj
Version	4cb1492b
Type	Smart Contract Platform
Platform	Java

Engagement Summary

Dates	October 23 - November 10, 2017
Method	Whitebox
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High Severity Issues	2	■■
Total Medium Severity Issues	3	■■■
Total Informational Severity Issues	2	■■
Total Undetermined Severity Issues	3	■■■
Total	10	

Category Breakdown

Denial of Service	6	■■■■■■
EVM incompatibility	3	■■■
Data Validation	1	■
Total	10	

Goals

The goal of the engagement was to evaluate the security of RSKj with a particular focus on potential denial of service vulnerabilities and errors in the EVM implementation.

Specifically, we sought answers to the following questions:

- **Bitcoin SPV proofs**: Are the merge-mining implementation and SPV proof verification correct? Two nodes should not disagree on the sequence of blocks in the blockchain.
- **Bridge**: Is the pre-compiled smart contract that implements the two-way peg correct? Malicious RSK contracts can interact with the Bridge to exploit denial of service or logical flaws that lead to incorrect accounting of ether or gas.
- **Consensus**: Is RSKj correctly handling block and signature validation? A miner could abuse consensus to receive more reward than intended.
- **EVM Compatibility**: Is the Ethereum EVM specification correctly followed? RSKj should execute contracts in the same manner as other Ethereum implementations. Inconsistencies may be exploited to cause denial of service or loss of ether.
- **JSON RPC**: Are the remote procedure calls to modules secure? Remote attackers may seek to abuse these interfaces to cause denial of service or execute commands without authorization.
- **Trie Implementation**: Is RSKj's Trie implementation correct? An adversary might be able to cause a block that exercises the Trie's worst case computational complexity, thus creating a denial of service.

Coverage

EVM compatibility. We analyzed the EVM implementation for errors in instruction execution and in gas consumption. We looked for arithmetic issues (*e.g.*, integer overflows), incorrect exception handling, and errors in the logic of the instructions. This area of concern was one of our primary targets. We found a number of severe vulnerabilities.

General DoS Attacks. RSK asked us to focus on denial of service issues. For that reason, we identified the six specific categories of bugs that are most likely to result in such a vulnerability:

Denial of Service Bug	Examples of What We Looked For	Number Discovered
Livelock	Infinite loops	1
Deadlock	Mutex locking and concurrency issues	0†
Memory Exhaustion	Memory leaks and incorrect resource cleanup	2
Algorithmic Complexity	Forcing hash collisions in data structure lookups	0
EVM Bugs	Contract operations that perform computation that is not commensurate with their gas cost	2
Data Corruption	Operations that can corrupt the node's internal state, causing it to be inoperable	1

† We found several instances of methods that could overwrite guarded variables without a lock on their associated mutex. However, we could not exploit these bugs.

Trie Implementation. We analyzed the Trie data structure for both logical errors and potential denial of service attacks (*e.g.*, algorithmic complexity vulnerabilities such as hash collision attacks), but were unable to discover any. Since the TrieStore holds serialized copies of Tries, we checked if there were any instances of unnecessary saving to the TrieStore, which would impose a significant memory overhead. All usage of the TrieStore appeared to be correct.

Due to time constraints, we did not cover the potential of a malicious or non-compliant miner. Further, RSK provided guidance that investigation into the RSKj wallet was a low priority. It was only briefly reviewed. The other areas of concern were either analyzed with a lower priority or were not reviewed.

Recommendations Summary

Short Term

Remove the memory leaks in the Trie. Close the input streams used in the Trie. This prevents a potential memory leak that could lead to denial of service.

Properly delete cached headers in the BlockStore. Use the correct arguments in the `remove` function. This resolves a memory leak that could potentially cause a denial of service or, worse yet, incorrect headers being returned.

Fix the erroneous EVM instruction implementations. Fix the CALL gas computation, use the correct value supplied to the CREATE instruction, and remove the duplicate log entries. The current CALL gas computation and CREATE instruction can be used to introduce backdoors in smart contracts. The duplicated log entries can cause a misinterpretation of events.

Fix the erroneous RLP encoding. Fix the encoding function or remove it if it is never used. This has the potential to corrupt internal state and deny service to users.

Implement the missing Ethereum Improvement Proposals (EIPs). Implement EIPs 160, 161, and 170. These serve to thwart denial of service attacks.

Remove or fix the ECKey.`verify` function. This function will cause an infinite recursion and ultimately a stack overflow exception. At a minimum, this code should be commented out because a future refactor or addition to the code might exercise it.

Long Term

Keep RSKj up to date with EthereumJ. RSKj is based on a fork of EthereumJ. Ensure that bugfixes and improvements made upstream are incorporated into RSKj on a timely basis.

Consider sponsoring development of the EthereumJ library. EthereumJ is a critical component of the RSKj solution. RSKj stands to benefit if consistent coding standards and security improvements are adopted by EthereumJ.

Document the EIPs implemented in RSKj. Ethereum is in constant evolution. Explicitly document which versions of the Ethereum specification RSKj has implemented, along with any deviations or incompatibilities with the specification.

Maintain syntactic consistency with the Yellow Paper and EIPs. Wherever possible, use similar variable names and code structure to match the specification that is being implemented. This will make deviations from the specification easier to identify.

Integrate static analysis tools and source code linters into the build process. Tools like Infer and Spotbugs can automatically detect many common errors at build time. Infer produces few false positives, and should be set to fail the build when errors are detected.

Use a differential fuzzer and improve the test coverage for the EVM instructions. Issues were found due to erroneous implementations of the Yellow Paper. Consider building a differential fuzzer to compare the executions of smart contracts on RSKj versus other Ethereum clients.

Findings Summary

#	Title	Type	Severity
1	Resource Leaks in Trie	Denial of Service	Undetermined
2	Headers Not Properly Deleted in the BlockStore	Denial of Service	Undetermined
3	Infinite Loop in EthereumJ Key Verification	Denial of Service	Informational
4	Integrate Infer into the RSKj Build Process	Denial of Service	Informational
5	Erroneous Gas Computation in CALL Breaks Sending Ether to a Contract	EVM incompatibility	High
6	Wrong msg.value Parameter in CREATE Leads to a Broken Contract	EVM incompatibility	High
7	Duplicated Logs May Lead to Misinterpreted Events	EVM incompatibility	Medium
8	Incorrect Gas Computation in Modexp	Denial of Service	Medium
9	Missing Implementation of EIPs May Lead to Denial of Service	Denial of Service	Medium
10	Incorrect Encoding Implementation Leads to Wrong RLP Encoding	Data Validation	Undetermined

1. Resource Leaks in Trie

Severity: Undetermined
Type: Denial of Service
Target: TrieImpl

Difficulty: Low
Finding ID: TOB-RSK-001

Description
The DataInputStreams used in message parsing (line 204 of TrieImpl.java),

```
ByteArrayInputStream bstream = new ByteArrayInputStream(message,
position, msglength);
DataInputStream istream = new DataInputStream(bstream);
```

deserialization (line 712),

```
ByteArrayInputStream bstream = new ByteArrayInputStream(bytes);
DataInputStream dstream = new DataInputStream(bstream);
```

and `getSerializedNodeLength` (line 788)

```
ByteArrayInputStream bstream = new ByteArrayInputStream(bytes, offset,
SERIALIZATION_HEADER_LENGTH);
DataInputStream dstream = new DataInputStream(bstream);
```

are never closed. Java requires that input streams be manually closed in order for their resources to be released and garbage collected. The severity of this flaw is “undetermined” because it would require a significant amount of effort to quantify the number of bytes that are leaked per message sent, if any. `ByteArrayInputStream`s do not require a call to close, but it is generally good practice to do so.

Exploit Scenario
The memory resources of the `DataInputStreams` may accumulate over time, even through normal, non-malicious use. However, an attacker can exacerbate this bug by causing the Trie to process more and/or larger inputs.

Recommendation
Wrap all usages of Java IOStreams in a `try/finally` block:

```
DataInputStream dstream = new DataInputStream(bstream);
try {
    /* do things with dstream */
} finally {
```

```
    dstream.close();
    bstream.close();
}
```

Alternatively, you can use the try-with-resources syntax introduced in Java 7:
https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

2. Headers Not Properly Deleted in the BlockStore

Severity: Undetermined
Type: Denial of Service
Target: BlockStore

Difficulty: Low
Finding ID: TOB-RSK-002

Description
In `co.rsk.net.BlockStore`, the function calls to `headersbynumber.remove(…)` and `headersbyparent.remove(…)` are expecting a `Long`, but in two instances within BlockStore are instead passed a Set. This will not cause an exception; it will silently fail to remove anything from the `headersbynumber` and `headersbyparent` maps. At best, this will just cause a memory leak. At worst, there might be a scenario in which old headers that were thought to have been removed could be erroneously retrieved later.

The offending lines in BlockStore.java are 240 and 248.

```
Set<ByteArrayWrapper> byNumber = this.headersbynumber.get(nkey);
if (byNumber != null) {
    byNumber.remove(key);
    if (byNumber.isEmpty()) {
        this.headersbynumber.remove(byNumber); ⬅
    }
}

Set<ByteArrayWrapper> byParent = this.headersbyparent.get(pkey);
if (byParent != null) {
    byParent.remove(key);
    if (byParent.isEmpty()) {
        this.headersbyparent.remove(byParent); ⬅
    }
}
```

Exploit Scenario
An attacker causes a significant number of blocks to be stored in the BlockStore. This induces a memory exhaustion denial of service. Trail of Bits has not evaluated whether such an attack vector is possible, since doing so would require a significant amount of effort and this bug is trivially fixable. Therefore, the severity of this bug has been classified “undetermined.”

Recommendation
Replace the arguments to `remove()` with `nkey` and `pkey`, respectively.

3. Infinite Loop in EthereumJ Key Verification

Severity: Informational
Type: Denial of Service
Target: ECKey

Difficulty: Low
Finding ID: TOB-RSK-003

Description
There are several overloads of the `verify(…)` function in `org.ethereum.crypto.ECKey`. The overload on line 697 with three byte array arguments will always recursively call itself with no modification to the arguments:

```
public static boolean verify(byte[] data, byte[] signature, byte[] pub) {
    return verify(data, signature, pub);
}
```

Calling this function will always result in an infinite recursion.

This erroneous function is called from the two argument overload on line 709. However, that version does not appear to be called from RSKj.

Exploit Scenario
If either of the aforementioned overloads of `verify(…)` are ever used within RSKj, it will cause an infinite loop resulting in a stack overflow exception.

Recommendation
In the short term, it should be safe to comment out those two overloads, since they do not appear to be used within ethereumj or RSKj.

Note that this issue has been fixed in the latest version of EthereumJ.

Ultimately, RSKj should pull in the latest changes from EthereumJ. EthereumJ's fix uses the function `decodeFromDER` (ECKey.java#L632) which is missing in RSKj, though, so this may not be a trivial upgrade.

4. Integrate Infer into the RSKj Build Process

Severity: Informational
Type: Denial of Service

Difficulty: N/A
Finding ID: TOB-RSK-004

Description
Infer can automatically detect null pointer exceptions, resource leaks, and certain types of concurrency and locking issues that can lead to the types of attack vectors which concern RSK. For example, deadlocks due to synchronization and mutex locking issues are difficult to test for in unit tests, since they often require specifically timed inputs that are hard to reproduce. Infer can catch many types of these bugs.

Exploit Scenario
An attacker submits multiple messages to the node, inducing a deadlock due to incorrect mutex locking, thus causing a denial of service. Infer can be used to preemptively detect such concurrency issues.

Recommendation
Run Infer and fix all the bugs that it reports.

Long term, integrate Infer into the build process and fail the build when it finds a bug. Infer is available as a Gradle plugin and can be added to `rskj-core/build.gradle`.

5. Erroneous Gas Computation in CALL Breaks Sending Ether to a Contract

Severity: High | Difficulty: Easy
Type: EVM incompatibility | Finding ID: TOB-RSK-005

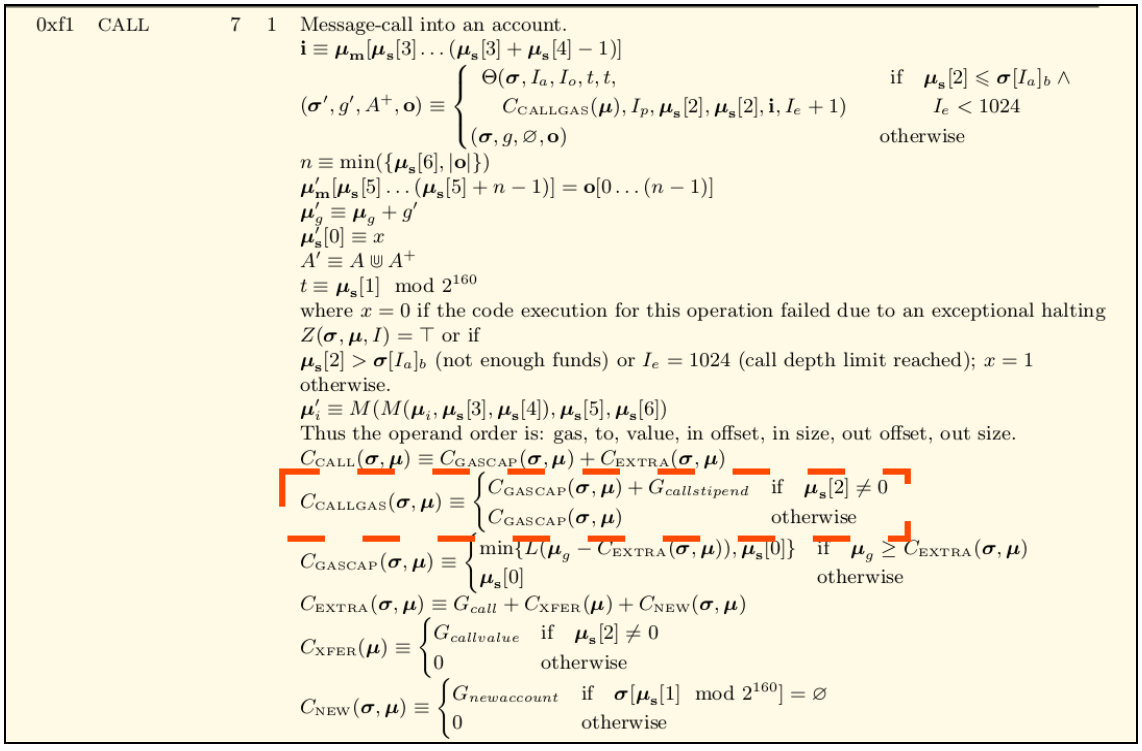
Description
During a call to another contract, an amount of gas should be provided to the callee.

The gas computation of CALL in RSKJ does not follow the yellow paper specification. In particular, it does not transfer the minimum required amount of gas during the transfer of ethers. As a result, it is not possible to send ethers to a contract.

As a consequence:

- An attacker can take advantage of this difference to create a smart contract which would be secure in ethereum but would be malicious in RSK.
- The bug can be triggered accidentally by RSK users, leading to unintended outcomes, such as trapped ether.
- If RSKJ allows another client to connect to the blockchain, the execution of the smart contract may lead to a different result, leading to a possible fork of the blockchain.

The Yellow Paper specifies that a call containing a non-zero `msg.value` supplies a minimum of a “stipend” amount of gas (2300) to the callee.



0xf1 CALL 7 1 Message-call into an account.

$$\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$$

$$(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{otherwise} \end{cases}$$

$$n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], |\mathbf{o}|\})$$

$$\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$$

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$$

$$\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$$

$$A' \equiv A \uplus A^+$$

$$t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}$$

where $x = 0$ if the code execution for this operation failed due to an exceptional halting $Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) = \top$ or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_a]_b$ (not enough funds) or $I_e = 1024$ (call depth limit reached); $x = 1$ otherwise.

$$\boldsymbol{\mu}'_i \equiv M(M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$$

Thus the operand order is: gas, to, value, in offset, in size, out offset, out size.

$$C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$$

$$C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$$

$$C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_g - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if} \quad \boldsymbol{\mu}_g \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$$

$$C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$$

$$C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$$

Figure 1: The yellow paper (EIP-150 revision) p.29.

This fee ensures that a called contract can at least perform some basic operations.
RSKJ does not follow this implementation (VM.java#L1260-L1377). The stipend is not added if the gas transferred is less than the remaining gas (VM.java#L1318-L1321).

This has a direct impact on the use of the `send` function. `addr.send(v)` is equivalent to `addr.gas(0).value(v)()`. As a result, the destination of `send` will have 0 gas instead of 2300. As a result, `send` will fail if the destination is a contract.

Appendix D contains a test case to trigger the issue.

Exploit Scenario
Bob creates a contract that allows users to store and withdraw some ethers. During the withdrawal, the contract sends some ethers to another contract as a fee. Alice stores 10 ethers in this contract. After some times she decides to withdraw the 10 ethers. However, due to the error in the CALL gas computation, the withdraw function fails and Alice can't withdraw the 10 ethers.

Appendix E illustrates this attack.

Recommendation
Fix the CALL gas computation.

We found other potential issues related to the CALL gas computation. We recommend rewriting the CALL gas computation entirely. The gas computation of complex instructions (CALL, CREATE, ...) needs to follow a structure closer to the Yellow Paper. Using the same variable names and operations order would allow easier verification of the correct behavior of these instructions.

Finally, create unit tests for greater coverage of the instructions' behavior.

6. Wrong msg.value Parameter in CREATE Leads to a Broken Contract

Severity: High
Type: EVM incompatibility

Difficulty: Easy
Finding ID: TOB-RSK-006

Description
Ether can be transferred during the creation of a contract. The RSKJ implementation does not use the value given as the parameter of the CREATE instruction, but uses instead the value of the current transaction. As a result, the created contract receives an incorrect amount of ether in `msg.value`.

Similar to TOB-RSK-005, the consequences of this difference are:

- An attacker can take advantage of this difference to create a smart contract which would be secure in ethereum but would be malicious in RSK.
- The bug can be triggered accidentally by RSK users.
- If RSKJ allows another client to connect to the blockchain, the execution of the smart contract may lead to a different outcome, leading to a possible fork of the blockchain.

In the function `createContract` (Program.java#625-627), `getCallValue()` is used instead of the `endowment` (Program.java#L572). As a result, the `msg.value` of the current transaction is supplied to the constructor instead of the one provided by the CREATE instruction.

```
ProgramInvoke programInvoke = programInvokeFactory.createProgramInvoke(
        this, new DataWord(newAddress), getOwnerAddress(), getCallValue(), gasLimit,
        newBalance, null, track, this.invoke.getBlockStore(), byTestingSuite());
```

This issue would make any contract creation fail if the caller were a function with a non-zero `msg.value` and the constructor were not `payable`.

Appendix D contains a test case to trigger the issue.

Exploit Scenario
Bob creates a contract that allows users to store and withdraw some ethers. The withdraw function creates a non-payable contract used as proof of withdrawing. Alice decides to store 10 ethers in the contract. After some times she decides to withdraw the 10 ethers. Due to the wrong value sent to the created contract in withdrawal, the creation of the contract and the call to withdraw fail. As a result, Alice is not able to retrieve the 10 ethers. Appendix E illustrates this attack.

Recommendation
Use the correct value supplied to CREATE.

Similar to TOB-RSK-005, create unit tests for greater code coverage to test the instructions' behavior.

7. Duplicated Logs May Lead to Misinterpreted Events

Severity: Medium
Type: EVM incompatibility
Difficulty: Easy
Finding ID: TOB-RSK-007

Description
The Ethereum blockchain allows events recording through log operations. These events can be used by light clients to receive information from the blockchain. In RSK, the logs emitted during the creation of a contract are duplicated. As a result, a client based on the blockchain events can be fooled.

The logs are added to the program result during the creation of a contract in `createContract` (Program.java#637):

```
getResult().merge(result);
```

`merge` is defined in ProgramResult#203-206

```
public void merge(ProgramResult another) {
        addInternalTransactions(another.getInternalTransactions());
        addDeleteAccounts(another.getDeleteAccounts());
        addLogInfos(another.getLogInfoList());
```

The logs are added a second time in `createContract` (Program.java#673):

```
getResult().addLogInfos(result.getLogInfoList());
```

As a result, any log in a constructor is duplicated.

Appendix D contains a test case to trigger the issue.

Exploit Scenario
Bob creates a contract which stores some investments. The investments are logged. During the creation of the contract, Bob made an initial investment. Due to the duplicated logs, Bob's investments is announced twice. As a result, Alice believes that Bob made more investments than he actually did.

Recommendation
Remove the duplicated logs.

Similar to TOB-RSK-005 and TOB-RSK-006, create unit tests for greater code coverage to test the instructions' behavior.

8. Incorrect Gas Computation in Modexp

Severity: Medium
Type: Denial of Service

Difficulty: High
Finding ID: TOB-RSK-008

Description
The Modexp precompiled contract (PrecompiledContracts.java#L233-L257) is supposed to cost a minimum of 400 gas. In reality, it only costs four times the number of bytes of its arguments.

```
@Override
public long getGasForData(byte[] data) {
    return data != null ? 4 * data.length : 400;
}
```

We suspect that the `4 * data.length` value was meant to be added to `400`.

Fortunately, Modexp will throw a Java exception causing the contract to consume all of the available gas if it is passed zero bytes of data. If that exception were not thrown, then a maliciously crafted call to Modexp would cost zero gas and create a denial of service vulnerability. Regardless of the exception, a carefully crafted call to Modexp will cost less gas than intended.

Exploit Scenario
Bob creates a contract with a significant number of Modexp calls, for example, by manually calling it by address. These calls will cost significantly less gas than was intended, and produce a load on the network that is not commensurate with the gas cost.

```
contract Test {
    function test() {
        address a = 0x1000007;
        a.call(.);
    }
}
```

Recommendation
The gas computation for Modexp has been completely rewritten in the latest version of EthereumJ, fixing the bug. Short term, we recommend pulling the latest version of `PrecompiledContracts.java` from EthereumJ.

Long term, unit tests should be created to check the intended gas consumption of precompiled contracts like Modexp.

9. Missing Implementation of EIPs May Lead to Denial of Service

Severity: Medium | Difficulty: Undetermined
Type: Denial of Service | Finding ID: TOB-RSK-009

Description
The Spurious Dragon hardfork led to the implementation of four new Ethereum improvements (EIP 155, 160, 161, 170). These improvements make denial of service attacks more difficult. RSK implements only EIP 155. As a result, RSK may be more vulnerable to denial of service attacks than the current Ethereum blockchain.

The missing EIPs are:

- EIP 160: EXP cost increase
- EIP 161: State trie clearing (invariant-preserving alternative)
- EIP 170: Contract code size limit

Note that we have not evaluated whether the vulnerabilities addressed by these EIPs are realistically exploitable in RSK.

Exploit Scenario
Bob creates a contract that extensively uses the EXP instruction. As a result, he is able to generate a lot of computation in the RSK nodes for a low price, leading to a slowdown of the transaction processing.

Recommendation
Implement the missing EIPs.

The Ethereum specification is constantly evolving. Be sure to clearly document which EIPs are implemented in RSK and which are not.

10. Incorrect Encoding Implementation Leads to Wrong RLP Encoding

Severity: Undetermined | Difficulty: Undetermined
Type: Data Validation | Finding ID: TOB-RSK-010

Description
A contract can be encoded with RLP encoding. The implementation of this encoding is not correct and does not store the expected information. As a result, any attempt to use this encoding may be incorrect.

We did not assess the impact of this issue, as we did not assess the database implementation.

The function `getEncoded` (ContractDetailsCacheImpl.java#L202) is meant to return the RLP representation of the `storage` and `bytesstorage` `HashMap` of the contract. The function is composed of three loops. The first loop saves to the `keys` and `values` arrays the `storage`.

```
for (DataWord key : storage.keySet()){

    DataWord value = storage.get(key);

    keys[i] = RLP.encodeElement(key.getData());
    values[i] = RLP.encodeElement(value.getNoLeadZeroesData());

    ++i;
 }
```

The first problem comes from the second loop, which saves `bytesStorage` to the `keys` and `values` arrays, removing the previously saved `storage` values.

```
for (DataWord key : bytesStorage.keySet()) {
    byte[] value = bytesStorage.get(key);

    keys[i] = RLP.encodeElement(key.getData());
    values[i] = RLP.encodeElement(value);

    ++i;
}
```

Moreover, `i` is not re-initialized, which may lead to an out of bound error. Note that the third loop saves `bytesStorage` again, but in `keys2` and `values2`.

```
for (DataWord key : bytesStorage.keySet()) {
    byte[] value = bytesStorage.get(key);

    keys2[i] = RLP.encodeElement(key.getData());
    values2[i] = RLP.encodeElement(value);

    ++i;
}
```

Finally, if `bytesStorage` is not empty `getEncoded` returns the same variables two times, instead of returning the rlp encoding of `keys2` and `values2`.

```
byte[] rlpKeysList2 = RLP.encodeList(keys2);
byte[] rlpValuesList2 = RLP.encodeList(values2);
return RLP.encodeList(rlpKeysList, rlpValuesList, rlpCode, rlpKeysList,
rlpValuesList);
```

Exploit Scenario
The RSK database of Alice is corrupted due to the wrong RLP encoding. As a result, she is not able to use the platform.

Recommendation
Fix the RLP encoding.

EthereumJ implemented a light version of this encoding in 6863351d, and the function is not anymore supported in the master branch (ContractDetailsCacheImpl.java#L96-L98). Consider investigating the latest modifications of EthereumJ.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Related to**
Access Controls	Authorization of users and assessment of rights
Auditing and Logging	Auditing of actions or logging of problems
Authentication	Identification of users
Configuration	Security configurations of servers, devices or software
Cryptography	Protecting the privacy or integrity of data
Data Exposure	Unintended exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	Causing system failure
Error Reporting	Reporting of error conditions in a secure fashion
EVM incompatibility	Compatibility with the ethereum specification
Patching	Keeping software up to date
Session Management	Identification of authenticated users
Timing	Race conditions, locking or order of operations
Undefined Behavior	Undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users' information is at risk, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. SpotBugs Documentation

Consider using Spotbugs. It produces several orders of magnitude more warnings than Infer and many of them are false positives. However, it can find Java-specific bugs that Infer cannot.

For example, Spotbugs could have detected TOB-RSK-002 by producing a warning at compile time that an incorrect argument type was being passed to `remove`. It is also able to detect many other exploitable bugs, such as certain types of infinite loops.

Spotbugs can be integrated into RSKj's build process by including it as a plugin inside `rskj-core/build.gradle`.

False positives can be filtered out by creating an `exclude_filter.xml` file and configuring Spotbugs to use it within the Gradle build file:

```
spotbugs {
    effort = 'max'
    excludeFilter = rootProject.file('exclude_filter.xml')
    ignoreFailures = false
    reportLevel = 'low'
    sourceSets = [sourceSets.main]
}
```

The format of the exclude filter is documented here:
http://spotbugs.readthedocs.io/en/latest/filter.html

C. Coding Practices

This section describes coding practices that encourage future bugs and obfuscate the codebase, as well as suggestions for how to mitigate their risks. At best, these practices frustrate code review and should be avoided for that reason alone.

Debugging and Unused Code

RSKj contains substantial code for debugging purposes or that is wholly unused. It is not trivial to determine the code intended for each use. Debugging and/or dead code increases the attack surface. It should be clearly identified and correctly removed in production. For example, it is easy to forget to remove classes like SamplePrecompiledContract.java and accidentally maintain this precompiled class in production.

Java Reflection

Java reflection is used to invoke precompiled contract methods. This can be dangerous. It introduces the risk of unintentionally allowing a contract to call a method from a precompiled contract that was never intended to be made public.

For example, reflection in the 2-way peg code allows any contract to call any function within `Bridge.java` and `RemascContract.java`. While this does not appear to be a security issue now, it would be easy for a developer to later add a function to these files that they believe is private but, in fact, becomes callable thanks to reflection. Further, because control flow is determined at runtime, reflection hides bugs from static analysis.

Carefully consider the need for Java reflection.

Java Coding Practices

Several common Java errors were identified, such as overriding `equals()` but not `hashCode()`, or using primitive `arrays` as Map keys. These have the potential to cause security issues. Because EthereumJ does not consistently follow good coding practices, RSKj inherits the flaws.

A `grep` of the codebase reveals twenty instances where a primitive byte array is used as a map key: `grep -r "Map<byte\[\]" **/*.java`. In Java, this is rarely desirable, because primitive byte arrays do not overload `equals()` and `hashCode()`. This means that a `Map<byte[],_>` can contain *multiple* keys containing the same byte sequence. Maintaining such code in the codebase might encourage future developers to improperly use this idiom. The following code provides an example of the danger:

```
byte[] k1 = new byte[]{1, 2, 3};
byte[] k2 = new byte[]{1, 2, 3};
Map<byte[],String> map = new HashMap<>();
map.put(k1, "foo");
map.put(k2, "bar");
System.out.println(map.size());
System.out.println(map.get(k1));
System.out.println(map.get(new byte[]{1, 2, 3}));
```

This code will print out "2⏎foo⏎null⏎", despite the fact that the programmer probably expected it to print out "1⏎bar⏎bar⏎".

We inspected each of the instances and found none to be vulnerable.

Style Guidelines

RSKj was not written with a consistent coding style. Consider choosing and enforcing a common style guide. This is not solely an aesthetic recommendation. Doing so will help increase intelligibility and actually help programmers catch common errors. For example, most modern style guides require including braces after all `if`, `for`, and `while` blocks, even if they are syntactically optional. This requirement would have made errors like the one that led to Apple's "goto fail" SSL bug more evident.

Loop Behavior

There are instances in the code where loops omit the first and/or last elements of a list. For example, line 517 of BridgeSupport.java starts at index one and ends at the second-to-last index:

```
for (int j = 1; j < chunkList.size() - 1; j++) {
```

Line 543 of the same file also skips the first entry:

```
for (int i = 1; i < chunks.size(); i++) {
```

In these cases where the Bridge code is tightly coupled and split across multiple files, it is particularly difficult to determine whether this behavior is correct. It is challenging for a programmer to statically determine the semantics of the lists being iterated in these contexts. Therefore, we recommend at a minimum adding comments to these lines explaining the unusual iteration bounds.

Test coverage

One of the strengths of the RSK architecture is that it provides easy-to-use unit tests at different levels (*e.g.,* from the EVM instruction level all the way up to the transaction level). However, we found that the unit tests lack sufficient code coverage, and RSKj would benefit from greater coverage. Many issues found in this audit could have been discovered with more testing.

Syntax of Specification Implementation

Implementing a specification (e.g., the Ethereum Yellow Paper and EIPs) is error-prone and necessitates careful reviews. We recommend that the implementation follow the syntax of the specification as closely as possible. For example, such a practice may have avoided TOB-RSK-005, an issue in the gas CALL computation.

Other Implementation Notes

The logger warnings on lines 372 and 373 of Bridge.java mention the wrong function name, which can cause confusion when analyzing logs. The function name should in fact be "`addSignature`":

```
logger.warn("Exception in releaseBtc", e);
throw new RuntimeException("Exception in releaseBtc", e);
```

D. Test cases

The following test case are based on TransactionTest.java.

```
public class TOBTransactionTest {

    private Transaction mycreateTx(Blockchain blockchain, ECKey sender, byte[]
receiveAddress,
                                   byte[] data, long value, long gasValue, long gasLimit) throws
InterruptedException {
        BigInteger nonce = blockchain.getRepository().getNonce(sender.getAddress());
        Transaction tx = new Transaction(
                ByteUtil.bigIntegerToBytes(nonce),
                ByteUtil.longToBytesNoLeadZeroes(gasValue),
                //ByteUtil.longToBytesNoLeadZeroes(3_000_000),
                ByteUtil.longToBytesNoLeadZeroes(gasLimit),
                receiveAddress,
                ByteUtil.longToBytesNoLeadZeroes(value),
                data);
        tx.sign(sender.getPrivKeyBytes());
        return tx;
    }

    private TransactionExecutor executeTransaction(Blockchain blockchain, Transaction tx) {
        Repository track = blockchain.getRepository().startTracking();
        TransactionExecutor executor = new TransactionExecutor(tx, new byte[32],
blockchain.getRepository(),
                blockchain.getBlockStore(), blockchain.getReceiptStore(), new
ProgramInvokeFactoryImpl(), blockchain.getBestBlock());

        executor.init();
        executor.execute();
        executor.go();
        executor.finalization();

        track.commit();
        return executor;
    }

    @Test
    public void TOB_RSK_6() throws IOException, InterruptedException {
```

```
        /*
        pragma solidity ^0.4.11;
        contract Bank{
           function () payable{
           }
        }

        contract BankTest{
           Bank bank;
           function BankTest(){
               bank = new Bank();
           }
           function test() payable{
             // send will fail as it reach the fallback function with 0 gas
             // instead of 2300
             require(bank.send(msg.value));
           }
        }
         */
        BigInteger nonce =
RskSystemProperties.CONFIG.getBlockchainConfig().getCommonConstants().getInitialNonc
e();
        Blockchain blockchain =
ImportLightTest.createBlockchain(GenesisLoader.loadGenesis(nonce,
                getClass().getResourceAsStream("/genesis/genesis-light.json"), false));

        ECKey sender =
ECKey.fromPrivate(Hex.decode("3ec771c31cac8c0dba77a69e503765701d3c2bb62435888d
4ffa38fed60c445c"));
        System.out.println("address: " + Hex.toHexString(sender.getAddress()));

        String code =
"6060604052341561000c57fe5b5b610015610071565b8090506040518091039060000f08015
1561002b57fe5b600060006101000a81548173ffffffffffffffffffffffffffffffffffffffff021916908373f
fffffffffffffffffffffffffffffffffffffff1602179055505b610080565b604051605280610164833901905
65b60d68061008e6000396000f300606060405260003 57c0100000000000000000000000
0000000000000000000000000000000000900463ffffffff168063f8a8fd6d14603a575bfe5b60406
042565b005b600060009054906101000a900473ffffffffffffffffffffffffffffffffffffffff1673ffffffffffff
ffffffffffffffffffffffffffff166108fc34908115029060405180905060006040518083038185888f19
350505050151560a75760006000fd5b5b5600a165627a7a723058203c7a8287a3fc250a21a3
977795b8503f06a26297768cd6cb6718c2775f0967b5002960606040523415600b57fe5b5b6
039806019600039600f30060606040525b600b5b5b565b0000a165627a7a7230582084c75
```

```
502cddbf559cb954d63c37b724a0207330da9eb8cb1b572ccccc04599770029";

        String abi =
"[{\"constant\":false,\"inputs\":[],\"name\":\"test\",\"outputs\":[],\"payable\":true,\"type\":\"fu
nction\"},{\"inputs\":[],\"payable\":false,\"type\":\"constructor\"}]\n";

        Transaction tx = mycreateTx(blockchain, sender, new byte[0], Hex.decode(code), 0x0,
1, 30000000);
        executeTransaction(blockchain, tx).getResult();
        byte[] contractAddress = tx.getContractAddress();

        // Build the data for the fallback function
        CallTransaction.Contract contract = new CallTransaction.Contract(abi);
        byte[] callData = contract.getByName("").encode();

        // call test_create, with msg.value == 0x1
        Transaction tx1 = mycreateTx(blockchain, sender, contractAddress, callData, 0x1, 1,
30000000);
        ProgramResult programResult1 = executeTransaction(blockchain, tx1).getResult();

        // Check that there is no exception
        Assert.assertEquals(null, programResult1.getException());
    }


    @Test
    public void TOB_RSK_6() throws IOException, InterruptedException {

    /*
        pragma solidity ^0.4.11;
         contract Called{
            function Called() {
            }
         }
         contract Test{
            function test_create() payable returns(bool){
              require(msg.value>0);
              // msg.value in the constructor should be 0
              // but it is msg.value of the caller
              // As a result msg.value > 0 in the constructor
              // and the construction fails as Called is not payable
              Called c = new Called();
            }
```

```
    }
    */

    BigInteger nonce =
RskSystemProperties.CONFIG.getBlockchainConfig().getCommonConstants().getInitialNonc
e();
    Blockchain blockchain =
ImportLightTest.createBlockchain(GenesisLoader.loadGenesis(nonce,
            getClass().getResourceAsStream("/genesis/genesis-light.json"), false));

    ECKey sender =
ECKey.fromPrivate(Hex.decode("3ec771c31cac8c0dba77a69e503765701d3c2bb62435888d
4ffa38fed60c445c"));
    System.out.println("address: " + Hex.toHexString(sender.getAddress()));

    String code =
"606060405341561000c57fe5b5b61011f8061001c6000396000f3006060604052600035 7c0
100000000000000000000000000000000000000000000000000000000900463ffffffff168063f
8a8fd6d1461003b575bfe5b61004361005d565b604051808215151515815260200191505060
0405180910390f35b600060006000341115156100715760006000fd5b610079610096565b80
905060405180910390600f080151561008f57fe5b90505b5090565b604051604e806100a68
339019056006060604052341560 0b57fe5b5b5b5b603380601b6000396000f300606060405
25bfe00a165627a7a7230582075e7f56bdd442a3535c7648128d36e20a4ffcbf532d510f93a26
67d5cf884d6f0029a165627a7a72305820166b215c8d54a51d7f42aa722d8d3af333855ea293
30d37fc6fe3dce7dba64d70029";

    String abi =
"[{\"constant\":false,\"inputs\":[],\"name\":\"test\",\"outputs\":[{\"name\":\"\",\"type\":\"bool\
"}],\"payable\":true,\"type\":\"function\"}]\n";

    Transaction tx = mycreateTx(blockchain, sender, new byte[0], Hex.decode(code), 0x0,
1, 30000000);//5*25036);
    executeTransaction(blockchain, tx).getResult();
    byte[] contractAddress = tx.getContractAddress();

    // Build the data for test_create()
    CallTransaction.Contract contract = new CallTransaction.Contract(abi);
    byte[] callData = contract.getByName("test").encode();

    // call test_create, with msg.value == 0x1
    Transaction tx1 = mycreateTx(blockchain, sender, contractAddress, callData, 0x1, 1,
30000000);
    ProgramResult programResult1 = executeTransaction(blockchain, tx1).getResult();
```

```
    // Check that there is no exception
    Assert.assertEquals(null, programResult1.getException());
}


@Test
public void TOB_RSK_7() throws IOException, InterruptedException {

/*
    contract LogContract{
        event Log();
        function LogContract(){
            Log();
        }
    }
    contract LOGTest{
        function test(){
            new LogContract();
        }
    }
    */

    BigInteger nonce =
RskSystemProperties.CONFIG.getBlockchainConfig().getCommonConstants().getInitialNonc
e();
    Blockchain blockchain =
ImportLightTest.createBlockchain(GenesisLoader.loadGenesis(nonce,
            getClass().getResourceAsStream("/genesis/genesis-light.json"), false));

    ECKey sender =
ECKey.fromPrivate(Hex.decode("3ec771c31cac8c0dba77a69e503765701d3c2bb62435888d
4ffa38fed60c445c"));
    System.out.println("address: " + Hex.toHexString(sender.getAddress()));

    String code =
"60606040523415600b57fe5b5b6101278061001b6000396000f300606060405260003 57c01
0000000000000000000000000000000000000000000000000000000900463ffffffff168063f8
a8fd6d1461003b575bfe5b341561004357fe5b61004b61004d565b005b61005561006f565b8
0905060405180910390600f080151561006b57fe5b505b565b604051607d8061007f833901
9056006060604052341560 0b57fe5b5b7f5e7df75d54e493185612379c616118a4c9ac802de
621b010c96f74d22df4b30a6040518090506040518091039 0a15b5b603380604a6000396000
f30060606040525bfe00a165627a7a72305820ca180ff8a4f48f0761ceccb547fa3dc04f4839ed
```

```
edec64c03dc4d03d626eca6e0029a165627a7a7230582043dd0b76a02e37af49b2f1d11da82
899cf862ddbfd78d22e21c55c159faa4cbd0029";

        String abi =
"[{\"constant\":false,\"inputs\":[],\"name\":\"test\",\"outputs\":[],\"payable\":false,\"type\":\"f
unction\"}]\n";

        Transaction tx = mycreateTx(blockchain, sender, new byte[0], Hex.decode(code), 0x0,
1, 30000000);//5*25036);
        executeTransaction(blockchain, tx).getResult();
        byte[] contractAddress = tx.getContractAddress();

        // Build the data for test()
        CallTransaction.Contract contract = new CallTransaction.Contract(abi);
        byte[] callData = contract.getByName("test").encode();

        // call test_create
        Transaction tx1 = mycreateTx(blockchain, sender, contractAddress, callData, 0x0, 1,
30000000);
        ProgramResult programResult1 = executeTransaction(blockchain, tx1).getResult();

        // Check that the value in the log is == 1
        LogInfo log = programResult1.getLogInfoList().get(0);

        // Check that only one log is present
        Assert.assertEquals(1, programResult1.getLogInfoList().size());
    }
}
```

E. Exploitable Smart Contracts

TOB-RSK-005

```
pragma solidity ^0.4.11;

contract Bank{

    mapping(address => uint) public contributions;
    address contractFee;

    function Bank(){
       // init contractFee
    }

    function contribute() payable{
        contributions[msg.sender] += msg.value;
    }

    // The send to contractFee will fail
    // Thereby withdraw() cannot be executed
    function withdraw(){
        require(contributions[msg.sender]>1 wei);
        require(contractFee.send(1 wei));
        contributions[msg.sender] -= 1 wei;
        require(msg.sender.send(contributions[msg.sender]));
        contributions[msg.sender] = 0 ;
    }
}
```

TOB-RSK-006

```
pragma solidity ^0.4.11;

contract Bank{

    mapping(address => uint) public contributions;
    function contribute() payable{
        contributions[msg.sender] += msg.value;
    }

    // The ProofOfWithdrawing creation will fail
    // Thereby withdraw() cannot be executed
    function withdraw() payable{
        require(msg.value >1 wei); // fee

        ProofOfWithdrawing p = new Proof(msg.sender)
        // store p somewhere

        msg.sender.send(contributions[msg.sender]);
        contributions[msg.sender] = 0 ;
    }
}
```

F. Fix Log

RSK made the following modifications to their codebase as a result of this report. Each of the fixes was verified by the audit team.

Finding 1: Resource Leaks in Trie
No immediate fix necessary because the underlying stream, a `ByteArrayInputStream`, does not leak resources.

Finding 2: Headers Not Properly Deleted in the BlockStore
Fixed by removing unused code.

Finding 3: Infinite Loop in EthereumJ Key Verification
Fixed by removing unused code.

Finding 4: Integrate Infer into the RSK build process
Informational only. No action required.

Finding 5: Erroneous Gas Computation in CALL Breaks Sending Ether to a Contract
https://github.com/rsksmart/rskj/pull/252

Finding 6: Wrong msg.value Parameter in CREATE Leads to a Broken Contract
https://github.com/rsksmart/rskj/pull/242

Finding 7: Duplicated Logs May Lead to Misinterpreted Events
https://github.com/rsksmart/rskj/pull/243

Finding 8: Incorrect Gas Computation in Modexp
Pulled in the latest version of Modexp from EthereumJ
https://github.com/rsksmart/rskj/pull/231

Finding 9: Missing Implementation of EIPs May Lead to Denial of Service
Two out of three recommendations addressed; one is still in progress.

9.1: EIP 160 implemented. https://github.com/rsksmart/rskj/pull/253
9.2: EIP 161 will not be implemented because RSK accounts have a significant creation cost.
9.3: EIP 170 implemented. https://github.com/rsksmart/rskj/pull/267

Finding 10: Incorrect Encoding Implementation Leads to Wrong RLP Encoding
Deprecated the faulty methods by throwing an exception when called.
https://github.com/rsksmart/rskj/pull/269

EIP 161 Discussion (Finding 9.2)
The vulnerability that EIP 161 addresses is insignificant in RSKj because the cost of creating an account in RSK is greater than in Ethereum. The following is a justification for this claim provided by RSK:

EIP 161 does not make sense in a platform that is protected from creating accounts with close to zero cost. EIP 161 was added to Ethereum because an attacker was able to create accounts at close to zero cost.

EIP 161 can only delete accounts in the following cases:

1. an empty account has zero value transferred to it through CALL;
2. an empty account has zero value transferred to it through SUICIDE;
3. an empty account has zero value transferred to it through a message-call transaction; or
4. an empty account has zero value transferred to it through a zero-gas-price fees transfer.

Case 1: Transferring zero value is only used in contract method calls. When a contract calls a contract/account that is non-existent, it will try to create an account for it, and CALL will return `true`. A contract can always check if the receiver exists as a contract with `EXTCODESIZE`, and in fact Solidity checks the existence of receiver contracts with `EXTCODESIZE` for each contract function call, so Case 1 never happens in practice.

Case 2: SUICIDE use is very uncommon because contracts have long lifetimes and there are low incentives to suicide a contract after use. A contract can be called more than once in a contract but will only transfer value if the contract balance is non-zero. Therefore, to suicide to multiple accounts, the contract must top up the balance after each suicide, which implies an additional cost. Also, suicide pays 25K gas to create a new account. Therefore, SUICIDE cannot be used currently to spam the state trie.

Case 3: A message-call transaction that transfers zero value costs. These transactions have a cost of at least 21K gas. A transfer of zero value modifies the source account because the nonce is incremented. It seems that the cost of creation of a destination account is not well considered in the original Ethereum design, because at least 25K gas should be paid, as CALL pays for the same operation. Therefore, CALLs are subsidized. However if an attacker wants to use this subsidy to spam the trie state, he can send one satoshi to each created address, and avoid EIP 161 account deletion. Therefore, account deletion in Case 3 does not solve any real problem.

Case 4: In RSK, this case occurs in REMASC reward sharing. But REMASC may pay zero at the beginning. However, miners do not currently rotate their coinbase addresses, so no spam is generated. In the future, REMASC will not pay to miners if the payment is below a threshold limit, based on the minimum gas price defined in the block header.

About Trail of Bits

Since 2012, Trail of Bits has helped secure some of the world's most targeted organizations and devices. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code.

Our clientele—ranging from Facebook to DARPA—lead their industries. Their dedicated security teams come to us for our foundational tools and deep expertise in reverse engineering, cryptography, virtualization, malware, and software exploits. According to their needs, we may audit their products or networks, consult on the modifications necessary for a secure deployment, or develop the features that close their security gaps.

After solving the problem at hand, we continue to refine our work in service to the deeper issues. The knowledge we gain from each engagement and research project further hones our tools and processes, and extends our software engineers' abilities. We believe the most meaningful security gains hide at the intersection of human intellect and computational power.