PegaSys Pantheon

Ethereum Client Security Assessment

December 5th, 2018

Prepared For:
Faisal Khan | *ConsenSys*
faisal.khan@consensys.net

Meredith Baxter | *ConsenSys*
meredith.baxter@consensys.net

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

Mike Myers | *Trail of Bits*
mike.myers@trailofbits.com

Paul Kehrer | *Trail of Bits*
paul.kehrer@trailofbits.com

Changelog:

October 22nd, 2018:	Initial report delivered
December 5th, 2018:	Updates and fixes

Executive Summary

From September 24th through October 19th 2018, PegaSys engaged with Trail of Bits to review the security of their Ethereum client, Pantheon. Trail of Bits conducted this assessment over the course of eight person-weeks with three engineers working from commit hash `68164f65cf7b0467cc5accf88c7c3f50cab9f568 [68164f65]` from the Pantheon repository.

In the first week, Trail of Bits reviewed the codebase at a high level, checked for known vulnerabilities in Pantheon's dependencies, reviewed the build process and the output of static analysis tools, and discussed Pantheon's usage of PRNG APIs, specifically Java's `SecureRandom`.

In the second week, we focused on the overall cryptographic design, the use of cryptographically secure pseudo-random number generation (CSPRNG), the selection of an entropy source for the CSPRNG, the local storage of private keys, and the risk from malformed public keys received from peer nodes. Many of these focal areas arose from discussions with ConsenSys at the start of the effort.

In week three, we focused on issues related to Pantheon's implementation of Ethereum's DevP2P "wire protocol," including its implementation of RLP deserialization (Recursive-Length Prefix encoding format, used by Ethereum network nodes). We also reviewed the EVM implementation, with a specific focus on potential denial-of-service attacks (*e.g.*, gas cost manipulation).

We focused week four on an examination of Pantheon's implementation of the Ethereum API specification, and its associated JSON-RPC-based interface. This included the integration of Pantheon with a custom test tool for Ethereum clients, Etheno, that uses differential analysis to help identify issues with a client's transaction handling.

Pantheon's codebase incorporates a robust set of unit tests that prevented many potential implementation errors. The Pantheon development team had good intuition about potentially problematic areas of their codebase, and had prepared well for this assessment.

One high-severity issue was discovered, related to publicly disclosed vulnerabilities in the version of one of Pantheon's Java package dependencies. The other findings were medium- and low-severity, with the typical impact being a potential denial-of-service. In the case of the unsecured JSON-RPC interface, the risk is partially mitigated by that interface being off-by-default.

PegaSys should integrate a dependency security-checking solution with the Pantheon build system. Our recommendations for the use of secure random number generation will eliminate the difficulties PegaSys has encountered (complexity, prediction resistance, and reseeding) without reducing security (*qq.v.* Appendices C and E). Pantheon must implement the recommended Host header check in its JSON-RPC interface in order to mitigate

browser-based attacks. Incorporating our additional unit tests for RLP and
EVM (*q.v.* Appendix D), including differential testing (*q.v.* Appendix F), will further reduce the likelihood of implementation errors.

Project Dashboard

Application Summary

Name	Pantheon
Version	`68164f65`
Type	Ethereum full-node client
Platform	Java

Engagement Summary

Dates	September 24 to October 19, 2018
Method	Whitebox
Consultants Engaged	3
Level of Effort	8 person-weeks

Vulnerability Summary

Total High-Severity Issues	1	■
Total Medium-Severity Issues	2	■■
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	5	■■■■■
Total Undetermined-Severity Issues	1	■
Total	12	

Category Breakdown

Access Controls	1	■
Cryptography	1	■
Data Exposure	1	■
Data Validation	4	■■■■
Patching	2	■■
Undefined Behavior	2	■■
Denial of Service	1	■
Total	12	

Engagement Goals & Coverage

During this assessment, Trail of Bits focused on Pantheon's use of cryptographic primitives, the correctness of its EVM implementation, any potential denial-of-service vectors, its implementation of the DevP2P and RLPx protocols, and its JSON-RPC API.

Crypto

- ✓ Provide guidance on using `SecureRandom` correctly
- ✓ Provide guidance on specifying JCE security providers
- ✓ Dynamic analysis crypto check with CryptoSense Analyzer
- ✓ Review public key exchange between nodes
- ✓ Provide guidance on node private key storage

EVM

- ✓ Evaluate the correctness of the EVM implementation
- ✓ Evaluate gas-cost calculations
- ✓ Identify any denial-of-service cases in the EVM implementation

DevP2P and RLPx

- ✓ Review DevP2P edge cases
- ✓ Review type-handling within RLP decoding implementation

JSON-RPC

- ✓ Review Pantheon's JSON-RPC method handlers for logic errors
- ✓ Review Pantheon's JSON-RPC method handlers for correctness
- ✓ Review the localhost-only assurance method for the JSON-RPC interface
- ✓ Investigate RPC edge cases

Misc.

- ✓ Static analysis check with DevSkim
- ✓ Enumerate dependencies and review associated codebases for important bugfixes
- ✓ Examine the use of JNI components RocksDB and Xerial Snappy-Java
- ✓ Examine the use of a Java Security Manager, if any, as a security sandbox

A future review may wish to examine Pantheon's implementation of the DevP2P peer discovery protocol, which is a remaining area to check for exceptional conditions that might result in a denial-of-service. Likewise, during this assessment we did not examine the use of the Trie data structure or whether there were any potential abuse cases that could cause a denial-of-service, because it was considered unlikely and not a current priority.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Adopt a dependency-security checking solution and integrate it into Gradle.** Manually checking the security alerts on every dependency in a project is inefficient and allows a longer window for vulnerabilities to be introduced. Integrating one of the open-source or commercial solutions for dependency-checking will alert the development team to dependency-related security issues as soon as is possible. (TOB-CPP-001)

❑ **Add a `Host` header check to the JSON-RPC HTTP interface.** Any HTTP interface designed to be localhost-only must check the `Host` header for requests to verify that they legitimately originate from localhost. Without this check, DNS rebinding attacks allow remote attackers to load JavaScript in the user's browser to query the JSON-RPC interface. (TOB-CPP-008)

❑ **Improve unit test coverage for RLP.** Consider adopting the additional RLP unit tests given in Appendix D. Testing for additional edge cases may prevent exceptions during RLP decoding. (TOB-CPP-009)

❑ **Fix any latent bugs related to edge cases in transaction handling.** See findings TOB-CPP-012 and TOB-CPP-013. Addressing these bugs will prevent a blockchain fork that may, in the worst case, result from a specially crafted transaction.

❑ **For the PRNG, turn on prediction resistance and stop re-seeding the RNG on every read.** The re-seeding behavior prevents correct use of the RNG on at least macOS, and appears to be related to the entropy-generation performance issues on AWS instances. (TOB-CPP-005)

❑ **Require encryption of the node private key on disk.** The private key is currently written to disk unencrypted, and could be easily read by other applications or captured in backups. Pantheon should require a password to derive a key using a password-based key-derivation function and use that key material to encrypt and authenticate the private key. (TOB-CPP-006)

Long Term

❑ **Switch to using `SecureRandom` directly.** The current CSRPNG implementation is overly complex and attempts to replicate features that are already available through the operating system's CSPRNG. Directly using `SecureRandom` will greatly simplify the code and reduce the risk of CSPRNG misuse. (TOB-CPP-005)

❑ **Improve integration test coverage for the JSON RPC interface.** Some edge cases appear to have been missed. See findings TOB-CPP-012 and TOB-CPP-013.

❑ **Join the two separate implementations of RLP decoding under one class, to assure consistency.** The `RLPInput` class hierarchy for performing a complete decoding is not fully consistent with the subset implementation of RLP in `RlpUtils`. See finding TOB-CPP-010.

❑ **Consider adopting differential fuzzing to ensure continued compatibility with other Ethereum clients.** Differential testing can find behavioral differences between Pantheon and other Ethereum clients, whether or not Pantheon is the more correct implementation. Unintended differences could cause a blockchain fork. See Appendix F.

❑ **Consider enabling the Java SecurityManager to minimize Pantheon's runtime privileges on the system to the minimum necessary.** In Appendix E we describe how to add the Java SecurityManager to a Java application. Use of the Java Security Manager is an effective defense-in-depth approach to reducing the impact of language- or logic-level exploits.

Findings Summary

#	Title	Type	Severity
1	Invalid entry set in key-value store due to object reuse	Undefined Behavior	Low
2	Multiple remote-code-execution CVEs in JSON deserialization package	Patching	High
3	Multiple CVEs in version of Jenkins server used for Pantheon project	Patching	Informational
4	Gas overflows can result in null pointer exceptions	Data Validation	Informational
5	Unnecessary complexity around setup and use of the CSPRNG	Cryptography	Low
6	Plaintext local storage of node private key risks disclosure	Data Exposure	Low
7	<removed after discussion with PegaSys>	n/a	n/a
8	Unsecured JSON-RPC interface	Access Controls	Medium
9	RLP decoding throws on encodings that report a length greater than Integer.MAX_VALUE	Data Validation	Informational
10	Implementation differences between RLP length calculation vs. decoding	Denial of Service	Medium
11	Pantheon permits RLP encoded ints with leading zeros	Data Validation	Informational
12	eth_getTransactionReceipt silently fails for raw transaction	Undefined Behavior	Undetermined
13	Inconsistent milestone defaults can lead to rejected transactions	Data Validation	Informational

1. Invalid entry set in key-value store due to object reuse

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-CPP-001
Target: `services/kvstore/src/main/java/net/consensys/pantheon/services/kvstore/InMemoryKeyValueStorage.java`

Description

The `entrySet()` method of `java.util.map` is allowed to successively return a single, mutable `Entry` object instance, overwriting the object's contents during each iteration. Therefore, the `HashSet` created on line 63 of `InMemoryKeyValueStorage.java` (*cf.* Figure 1.1) can potentially contain multiple copies of the same `Entry` object with contents equal to the last entry returned from `hashValueStore.entrySet()`.

```
58 @Override
59 public Stream<Entry> entries() {
60   Lock lock = rwLock.readLock();
61   try {
62     lock.lock();
63     return new HashSet<>(hashValueStore.entrySet())
64         .stream()
65         .map(e -> Entry.create(e.getKey(), e.getValue()));
66   } finally {
67     lock.unlock();
68   }
69 }
```

Figure 1.1: *Object reuse bug in the use of `Map.entrySet()`.*

This behavior is prohibited according to the Java Set interface API:

> *Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.*

The severity of this finding is classified as Low because this bug is dependent on the JVM's implementation of the underlying `Map` type and may not be a vulnerability in all deployment scenarios. Moreover, the code currently only appears to be used within tests.

Exploit Scenario
A unit test silently fails to exercise the desired case because the `InMemoryKeyValueStorage` instance discards all but the last entry added.

Recommendation
It appears as if the “`new HashSet`” is superfluous and can simply be removed to resolve this issue, since the elements of `hashValueStore` are cloned in the `map`. In the short term, confirm whether this fix is sufficient.

In the long term, add source code comments to avoid this issue in other areas of the code.

2. Multiple remote-code-execution CVEs in JSON deserialization package

Severity: High
Difficulty: Low
Type: Patching
Finding ID: TOB-CPP-002
Target: `ethereum/jsonrpc/src/main/java/net/consensys/pantheon/ethereum/jsonrpc/internal/parameters/JsonRpcParameter.java`
(which is, in turn, used from multiple other locations)

Description

There are multiple arbitrary code execution vulnerabilities in the version of the JSON deserialization component used by Pantheon. FasterXML jackson-databind before 2.8.11.1 and 2.9.x before 2.9.5 allows unauthenticated remote code execution because of an incomplete fix for the CVE-2017-7525 deserialization flaw. Pantheon uses jackson-databind 2.9.0. The associated utility class in Pantheon that uses the vulnerable dependency may be exposed to exploitation via multiple vectors: DevP2P, RLPx, or local JSON-RPC interfaces.

Dependency	Referenced In	Vulnerabilities
jackson-databind-2.9.0.jar	pantheon:default pantheon:runtime pantheon:compile pantheon:runtimeClasspath	CVE-2017-15095 CVE-2018-5968 CVE-2018-7489

Exploit Scenario

The above-mentioned JSON deserialization vulnerabilities could be exploited by an attacker able to send JSON input data to the `readValue` method of the `ObjectMapper`, as abstracted by the Pantheon class `net.consensys.pantheon.ethereum.jsonrpc.internal.parameters.JsonRpcParameter`. A successful exploitation would result in arbitrary code execution on the host running Pantheon, allowing an attacker to read Pantheon's stored private keys and/or issue transactions that steal funds.

Recommendation

To protect against these publicly known vulnerabilities, immediately transition to the current version of the Java package com.fasterxml.jackson.databind: 2.9.7 at the time of this writing.

Afterward, adopt a dependency-checking solution to automate the monitoring and alerting of dependencies for upstream security issues. One solution is to add the DependencyCheck plugin for Gradle, and run it via:

```
./gradlew dependencyCheckAnalyze
```

Then, check for its report in:

```
pantheon/build/reports/dependency-check-report.html
```

An alternative solution for automating dependency risk-checking is using OWASP Dependency Check (available as a Jenkins plugin) or Snyk for Java, each of which can automatically identify open-source dependencies and determine if there are any known (publicly disclosed) vulnerabilities.

References

- FasterXML/jackson-databind Home Page (FasterXML)
- GitHub issue discussing the CVE-2018-7489 problem and the fix

3. Multiple CVEs in version of Jenkins server used for Pantheon project

Severity: Informational Difficulty: Low
Type: Patching Finding ID: TOB-CPP-003
Target: Jenkins CI server at `http://forge-jenkins.kellstrand.com:8080/`

Description
There are multiple CVEs in Jenkins 2.137, 2.132 and earlier. PegaSys Pantheon is using a privately hosted Jenkins CI server that uses version 2.107.3.

This finding is listed as `Informational` severity because it is an incidental finding outside the scope of the Pantheon codebase assessment.

Exploit Scenario
There are quite a few vulnerabilities, the worst of which is that an unauthenticated user providing malicious login credentials could grant themselves administrator access to the Jenkins server. The impact to Pantheon should be limited: the integrity of the codebase on GitHub should be unaffected, and the Pantheon source code is already planned for an open-source release. An attacker might employ a denial-of-service of the project's CI testing, or attempt to move laterally with their access (*e.g.*, by attacking visitors to the Jenkins server or attempting credential re-use).

Recommendation
Update the version of Jenkins CI from version 2.107.3 to version 2.121.3 (released August 15th, 2018).

Then, subscribe to the jenkinsci-advisories Google Group or RSS feed to receive timely notifications on security updates.

References

- CVE-2018-1999001, CVE-2018-1999002, CVE-2018-1999003, CVE-2018-1999004, CVE-2018-1999005, CVE-2018-1999006, and CVE-2018-1999007: https://jenkins.io/security/advisory/2018-07-18/
- CVE-2018-1999042, CVE-2018-1999043, CVE-2018-1999044, CVE-2018-1999045, CVE-2018-1999046, CVE-2018-1999047: https://jenkins.io/security/advisory/2018-08-15/

4. Gas overflows can result in null pointer exceptions

Severity: Informational | Difficulty: Low
Type: Data Validation | Finding ID: TOB-CPP-004
Target: `ethereum/code/src/main/java/net/consensys/pantheon/ethereum/vm/EVM.java`

Description

The function for calculating gas cost returns `null` if the gas calculation overflows.

```
private Gas calculateGasCost(MessageFrame frame) {
  // Calculate the cost if, and only if, we are not halting as a result of a stack
underflow, as
  // the operation may need all its stack items to calculate gas.
  // This is how existing EVM implementations behave.
  if (!frame.getExceptionalHaltReasons().contains(INSUFFICIENT_STACK_ITEMS)) {
    try {
      return frame.getCurrentOperation().cost(frame);
    } catch (IllegalArgumentException e) {
      // TODO: Figure out a better way to handle gas overflows.
    }
  }
  return null;
}
```

Figure 4.1: *Gas cost calculation returns `null` on overflow.*

This is fine from an EVM compatibility perspective, because other implementations do not raise an exception on gas overflow and rather silently fail. However, the result of this function is passed to an instance of an `OperationTracer`. Currently, if the `DebugOperationTracer` is used, this will result in an uncaught null pointer exception when the gas cost is retrieved (*e.g.*, during logging).

The severity of this finding was classified "Informational" because it appears as if the only way this bug can manifest is if the system is run with debugging turned on, which should never happen in production.

Exploit Scenario

The system running with a `DebugOperationTracer` processes a transaction that overflows its gas cost calculation, causing an uncaught null pointer exception.

Recommendation

In the short term, document all uses of `calculateGasCost` to memorialize the fact that it can return a `null`. In the long term, devise a better way to handle gas overflows.

5. Unnecessary complexity around setup and use of the CSPRNG

Severity: Low
Difficulty: n/a
Type: Cryptography
Finding ID: TOB-CPP-005
Target: `/crypto/src/main/java/net/consensys/pantheon/crypto/*`

Description
The current codebase uses multiple CSPRNGs that subclass Bouncy Castle's DRBG implementation, and implement a custom re-seeding mechanism. The rationale for this is to have different security domains per CSPRNG and generate randomness such that an attacker who breaks one CSPRNG will not compromise the others. Each instance, however, uses the same algorithm, is seeded via the system CSPRNG, and then is re-seeded via calls to `nanoTime` to gain small quantities of entropy and derive some prediction resistance.

This edifice is large and fragile and attempts to derive a defense against a state-level actor where the attacker can modify `/dev/urandom` output, but can't read memory or insert malicious code. The approach drastically increases the implementation complexity for limited gain, requires constant vigilance to ensure the "correct" CSPRNG is used for its stated purpose, and introduces new potential points of failure.

Building a tiered hierarchy of CSPRNGs that feed into each other and attempting to separate them into security domains doesn't add real security. No significant advantage is derived against a real-world threat actor, and yet it makes comprehension of the system much more difficult for developers. Additionally, the probability of misuse of an CSPRNG outside its permitted security domain seems higher in the medium- to long-term, which would counteract the (limited) hypothetical advantage.

Exploit Scenario
The way these CSPRNGs are initialized is from a parent CSPRNG, which we are implicitly considering out of scope. If you posit that these CSPRNGs (for any reason) are structurally weak, then if the parent CSPRNG is weak, the quantity of entropy the child CSPRNGs work with is much lower than expected. If you assume the `DRBG` algorithm used to seed the CSPRNG is compromised, then all security domains would be compromised. Separation of randomness again confers no benefit.

Recommendation
Consensys should either use `SecureRandom` directly or use a singleton instance of the Bouncy Castle NIST SP800-90Ar1 `HASH_DRBG` random number generator with prediction resistance turned **on**, no personalization (which provides no security benefit here), and remove the custom prediction resistance mechanism (the subclass to do `nanoTime` re-seeds).

The best CSPRNG option available in Java is the `NativePRNG` (or `Windows-PRNG` on Windows). This is automatically selected by calling new `SecureRandom()` on a typical Java install and will provide good random data on *nix/BSD/macOS when calling nextBytes. You can ensure the selection of this even on systems without the default Java 8 `java.security` configuration by passing `-Djava.security.egd=file:/dev/urandom` and/or using `SecureRandom.getInstanceStrong`. On Windows, the best you can do in Java is to use `CryptGenRandom` to seed the SHA1PRNG.

Since the native form of `SecureRandom` is tied to the underlying operating system, then you may see significantly different performance characteristics depending on the version of the kernel. On older Linux kernels (2.x, 3.x) the CSPRNG behind `/dev/urandom` could sometimes be relatively slow (but still ~2MB/sec, more than enough for Pantheon) compared to Java's `SHA1PRNG`. On Windows, Java seeds the `SHA1PRNG` using `CryptGenRandom`, but can't exclusively use `CryptGenRandom` as its entropy source simply because the JDK lacks support for that. Because it just uses the `SHA1PRNG` on Windows, though, performance should be much higher than 2MB/sec.

If performance was a motivating factor for the current CSPRNG design choices, consider testing with JDK10 DRBGs, which are sufficiently performant. See Appendix C.

References

- The right way to use SecureRandom
- Myths about urandom
- Cryptographic Right Answers
- Challenges with randomness in multi-tenant Linux container platforms
- NIST Recommendations for RNGs Using Deterministic Random Bit Generators
- Cryptographically Secure Pseudo-Random Number Generator

6. Plaintext local storage of node private key risks disclosure

Severity: Low
Difficulty: Low
Type: Data Exposure
Finding ID: TOB-CPP-006
Target:
`/pantheon/src/main/java/net/consensys/pantheon/controller/KeyPairUtil.java`

Description
As noted in the previous ConsenSys code review, Pantheon currently stores its node's private key in plaintext, on a file on the local filesystem.

```
public static SECP256K1.KeyPair loadKeyPair(final Path home) throws IOException {
    final File keyFile = home.resolve("key").toFile();
    final SECP256K1.KeyPair key;
    if (keyFile.exists()) {
      key = SECP256K1.KeyPair.load(keyFile);
      LOGGER.info(
          "Loaded key {} from {}", key.getPublicKey().toString(), keyFile.getAbsolutePath());
    }
```

Figure 6.1: *key loaded from plaintext file. Excerpt from KeyPairUtil.java.*

Exploit Scenario
Each Ethereum node is expected to maintain a static private key which is saved and restored between sessions. This key is used during the ECIES (Elliptic Curve Integrated Encryption Scheme) handshake part of the RLPx protocol with other nodes, in order to exchange the AES key that protects their network session.

An attacker that gains access to the filesystem or backups that contain the configuration could directly access the stored private key. An attacker with a node's private key could decrypt captured traffic to/from that node, or spoof Ethereum messages as that node. This private key is not the same as the one used to sign transactions, so there is no direct risk of a theft of funds. Nevertheless, a compromised node key would re-enable denial-of-service attacks that the authenticated encryption of the DevP2P protocol was intended to protect against.

Recommendation
Encrypt private keys via an authenticated encryption scheme (`AES-GCM` or `ChaCha20Poly1305`) and derive the key used to encrypt via a password KDF like `scrypt`, `argon2id`, or `bcrypt`. Java crypto providers like Bouncy Castle implement key stores with password-based encryption, but avoid the default JKS Java keystores which are weak and easily cracked.

In the longer term, you could also provide an option for storing keys entirely inside HSMs via a PKCS11 JCE provider, or a cloud-based key management system via JCE providers. There are two kinds of cloud crypto services available: Key Broker or Key Management

Services (KMS), and Cloud HSMs. However, at the time of this writing, Microsoft's Azure KeyVault appears to be the only KMS that offers the `SECP256K1` support that Pantheon requires.

References

- Cryptographic Right Answers (*c.f.*, "password handling")
- Java PCKS#11 Reference Guide
- Bouncy Castle Keystore Security
- "Nail in the Java Key Store Coffin", PoC || GTFO 0x15
- Cloud Service Provider (CSP) Cloud Key Management Services (KMS)
    - AWS KMS and Supported Operations
    - GCP KMS
    - Microsoft Azure KeyVault and Supported Operations
- Cloud HSMs
    - AWS CloudHSM and Supported Operations
    - Microsoft Azure KeyVault (HSM backed mode)
    - Gemalto Cloud HSM
- DevP2P protocol's use of public keys for node identity

7. <removed after discussion with PegaSys>

Note: this issue was an apparent unhandled exception with regard to how Pantheon accepts malformed public keys. Upon further inspection and a discussion with PegaSys, it was determined that the exception is in fact handled in production, and the issue was removed from the report.

8. Unsecured JSON-RPC interface

Severity: Medium
Difficulty: Low
Type: Access Controls
Finding ID: TOB-CPP-008
Target:
`ethereum/jsonrpc/src/main/java/net/consensys/pantheon/ethereum/jsonrpc/JsonRpcHttpService.java`

Description
The JSON-RPC service (disabled by default when running `./pantheon`) is an unauthenticated interface. If the JSON-RPC service is activated, then the client is vulnerable to a DNS rebinding attack.

Exploit Scenario
An attacker tricks the user into loading a malicious website. This website loads various subdomains that (with the aid of DNS cache expiry) eventually results in JavaScript being loaded in the browser that can send requests to `127.0.0.1`. Since the JSON-RPC interface is unauthenticated, the attacker can now control the service.

Recommendation
Whitelist `localhost` as a `Host` header, and reject communication from any client that can't set that header. DNS rebinding relies on the ability to set an arbitrary FQDN to `127.0.0.1` so this mitigation prevents browser-based attacks.

References
- How your ethereum can be stolen through DNS rebinding
- Project Zero: agent rpc auth mechanism vulnerable to dns rebinding

9. RLP decoding throws on encodings that report a length greater than Integer.MAX_VALUE

Severity: Informational | Difficulty: Low
Type: Data Validation | Finding ID: TOB-CPP-009
Target: `ethereum/rlp/src/main/java/net/consensys/pantheon/ethereum/rlp/RLP.java`

Description
The Pantheon RLP implementation uses Java's signed integers for decoding, which means that any RLP string, byte array, or list that reports to be larger than 2^{31}-1 will cause an integer overflow, regardless of whether the encoding is actually valid.

The RLP specification allows for lengths of up to 256^8. Given that this is an impractically large size that is unlikely to fit in the memory of today's systems, some Ethereum clients implement their own lower limits for encoded length — 2^{64} seems to be a common choice.

Exploit Scenario
Alice sends the string "`\xBC\x01\x00\x00\x00\x00`" to Pantheon, which reports to be a string of length 2^{32}. This will cause `RLP.input` to throw an exception due to integer overflow. It is worth noting that although not inline with the protocol specification and not *clearly* intentional, discarding huge messages is a reasonable action. This issue is listed as only Informational severity.

Recommendation
In the short term, modify the error message and source code comments to indicate more explicitly the choice for Pantheon to implement a max length of 2^{31}-1 rather than the max length allowed by the RLP specification. Consider including the additional RLP unit tests given in Appendix D.

In the long term, ensure that the RLP implementation is consistent with other Ethereum clients, using differential testing.

References
- Ethereum wiki: RLP decoding
- RLP implementations that allow lengths up to the full 256^8
  - pyrlp RLP serialization library
  - ruby-RLP library
- RLP implementations where the maximum allowed length is 2^{64}
  - EthereumJ
  - Ethminer

10. Implementation differences between RLP length calculation vs. decoding

Severity: Medium
Difficulty: Low
Type: Denial of Service
Finding ID: TOB-CPP-010
Target: `ethereum/rlp/src/main/java/net/consensys/pantheon/ethereum/rlp/RLP.java` and `RlpUtils.java`

Description

There are two separate implementations of RLP decoding, one in the `RLPInput` class hierarchy for doing a complete decoding, and another independent implementation in `RlpUtils` for decoding only the offsets and lengths of RLP encoded elements in a byte stream. These implementations do not agree with each other. For example,

```
RlpUtils.decodeLength(h("0xbc0100000000").extractArray(), 0);
```

returns the length 6 for this incomplete RLP encoding, while

```
RLPInput in = RLP.input(h("0xbc0100000000"));
```

raises an `ArithmeticException` due to integer overflow for the exact same input.

Likewise,

```
RLP.decode(BytesValue.wrap(
             new byte[]{(byte)0xBC, 0x01, 0x00, 0x00, 0x00, 0x00}
            ))
```

throws an `ArithmeticException` due to integer overflow.

This is because the implementation in `RlpUtils` uses integer arithmetic and does not check for overflow. Therefore, any RLP encoding that reports a string, byte array, or list length greater than `Integer.MAX_VALUE` will cause `RlpUtils` to silently overflow, return an incorrect value, and fail to check whether the input is actually valid. `RLPInput` will not process such encodings either, but fails with different behavior.

In order to be compatible with RLP, both implementations must support parsing strings, byte arrays, and lists of up to length 256^8.

Exploit Scenario
Alice sends an RLP encoded string whose length is larger than `Integer.MAX_VALUE`. This causes `RlpUtils` to incorrectly segment the RLP stream, causing the payload of the string to be parsed as the second RLP entry.

This inconsistency between length precalculation and actual decoding can lead to a class of vulnerabilities with real-world consequences.

Recommendation
In the short term, ensure that both implementations have consistent behavior, and implement the additional RLP unit tests given in Appendix D. In the long term, settle on a single implementation that can perform both functions.

11. Pantheon permits RLP encoded ints with leading zeros

Severity: Informational Difficulty: Low
Type: Data Validation Finding ID: TOB-CPP-011
Target: `ethereum/rlp/src/main/java/net/consensys/pantheon/ethereum/rlp/RLP.java`

Description
In `RLP.decode` (one of the two code locations in Pantheon that decodes RLP-encoded integers), it is possible to take an encoded scalar. This method assumes a fixed-length sequence, and will decode an RLP-encoded integer even if it has leading zeros:

```
RLPInput in = RLP.input(h("0x0000D0"));
RLP.decode(in.raw());
```

The standard explicitly forbids this case for encoded scalars, and other RLP parsers throw an exception on it.

> "*...positive RLP integers must be represented in big endian binary form with no leading zeroes [sic] (thus making the integer value zero be equivalent to the empty byte array).* ***Deserialised positive integers with leading zeroes [sic] must be treated as invalid.***" (emphasis added)

This finding has "informational" severity because although RLP.decode makes this assumption, it is used only in test, not in production. All production RLP encoding and decoding methods in Pantheon explicitly specify whether the input is a fixed-length sequence or a scalar value.

Exploit Scenario
A new developer on Pantheon uses this implementation to decode RLP scalars, rather than the one intended for production. Pantheon processes a transaction with a positive integer erroneously encoded with leading zeros. Pantheon will ignore the error and accept the transaction, while other Ethereum clients would have rejected it, leading to a fork.

Recommendation
In the short term, ensure that Pantheon's RLP implementations are clearly documented as being for test-only or for production. Implement the additional RLP unit tests given in Appendix D.

In the long term, consider de-duplicating the RLP decoding methods so that it is not possible to use the "wrong" one in production. Ensure that the RLP implementation strictly adheres to the standard, *e.g.*, by performing differential testing against other RLP implementations.

12. eth_getTransactionReceipt silently fails for raw transaction

Severity: Undetermined | Difficulty: Low
Type: Undefined Behavior | Finding ID: TOB-CPP-012
Target: `EthGetTransactionReceipt.java`

Description

The `eth_getTransactionReceipt` JSON RPC call will return an invalid result. For example, a Pantheon instance was started using this command:

```
./gradlew run -Ppantheon.run.args="--no-discovery --datadir=/tmp/pantheontmp
       --miner-enabled --rpc-enabled --miner-coinbase
       fe3b557e8fb62b89f4916b721be55ceb828dbd73 --rpc-listen=127.0.0.1:1234
       --p2p-listen=127.0.0.1:33333
       --genesis=ethereum/core/src/main/resources/dev.json"
```

The following transaction was then submitted to Pantheon for mining:

```
{'from': '0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73', 'gas': '0x99999',
'gasPrice': '0x430e23400', 'value': '0x0', 'data':
'0x608060405234801561001057600080fd5b506101c0806100206000396000f3006080604052600
436106100575760003570100000000000000000000000000000000000000000000000000000009
00463ffffffff168063554c5abd1461005c5780638c9670b51461008b5780639ccb138f146101075
75b600080fd5b348015610068576000080fd5b50610071610181565b60405180821515151581526020
0191505060405180910390f35b34801561009757600080fd5b5061010560048036038101908080 3
515159060200190929190803590602001909291908035906020019082018035906020019080806 02
00260200160405190810160405280939291908181526020018383602002808284378201915050505
05050919291929050505061018a565b005b34801561011357600080fd5b5061017f600480360381 0
190808035906020019092919080359060200190929190803590602001908201803590602001908 08
06020026020016040519081016040528093929190818152602001838360200280828437820191505
05050505091929192905050505061018f565b005b60006001905090565b505050565b5050505600a16
5627a7a72305820c33d6d41fb62e921093df0df9278328c3f1f256bac6be1400b47d233c6b1aeff0
029', 'nonce': 0}
```

This transaction creates a contract. It is manually signed and submitted to Pantheon using `eth_sendRawTransaction`:

```
{'id': 1, 'jsonrpc': '2.0', 'method': 'eth_sendRawTransaction', 'params':
['0xf9023380850430e23400830999998080b901e0608060405234801561001057600080fd5b5061
01c0806100206000396000f300608060405260043610610057576000357c01000000000000000000
00000000000000000000000000000000000000900463ffffffff168063554c5abd1461005c578063
8c9670b51461008b5780639ccb138f14610107575b600080fd5b348015610068576000080fd5b5061
0071610181565b60405180821515151515815260200191505060405180910390f35b34801561009757
600080fd5b506101056004803603810190808035151590602001909291908035906020019092919080
803590602001908201803590602001908080602002602001604051908101604052809392919081815
2602001838360200280828437820191505050505050919291929050505061018a565b005b348015
61011357600080fd5b5061017f600480360381019080803590602001909291908035906020019092
919080359060200190820180359060200190808060200260200160405190810160405280939291908
1815260200183836020028082843782019150505050505091929192905050506101 8f565b005b60
006001905090565b505050565b5050505600a165627a7a72305820c33d6d41fb62e921093df0df92
78328c3f1f256bac6be1400b47d233c6b1aeff00291ca051bf58218652a7b0c4323c0b4af2f73860
28556b4695226fc18d99ff2569aaa9a07c117b21247c1d2fb19c643fc0d373e71a12a624d2831516
cf6057e9ea8dcf48']}
```

Pantheon then proceeds to mine the transaction and create the contract:

```
Successful creation of contract 0x42699a7612a82f1d9c36148af9c77354759b210b with
code of size 448 (Gas remaining: 488970)
```

However, subsequent calls to `eth_getTransactionReceipt` on the transaction hash return an invalid response, in which the `result` field is the transaction hash:

```
{'id': 1, 'jsonrpc': '2.0', 'result':
'0xbba27352c4f655a15fc9d85bc79166b13592528063642b6e95c9a74f2c9bcbcf'}
```

The severity of this finding is undetermined because it is unclear whether this is simply a bug in the JSON RPC interface or whether it is a manifestation of a more serious bug related to mining.

Furthermore, Lucas Saldanha investigated this bug on a different version of the codebase (RC2) and was unable to reproduce it, so it may be specific to the assessed version of the codebase (`68164f65`).

Exploit Scenario
This bug is a manifestation of a mining error that can result in a fork.

Recommendation
In the short term, determine the underlying cause of this bug and fix it. In the long term, add more integration tests for the JSON RPC interface, and regularly compare Pantheon to other Ethereum client implementations using a differential tester like Etheno (see Appendix F).

13. Inconsistent milestone defaults can lead to rejected transactions

Severity: Informational Difficulty: Low
Type: Data Validation Finding ID: TOB-CPP-013
Target: `ethereum/core/src/main/java/net/consensys/pantheon/ethereum/mainnet/MainnetTransactionValidator.java`

Description
Pantheon's `MainnetTransactionValidator` will raise a `REPLAY_PROTECTED_SIGNATURES_NOT_SUPPORTED` exception if it does not have a `chainId` specified but a transaction *does* have an explicit `chainId`. However, when using Pantheon with a custom genesis file and chain ID, any transaction with a chain ID specified—even if it is the *correct* chain ID—will be rejected by Pantheon. This appears to be due to the fact that Pantheon will default to Frontier milestones (which do not include transaction replay protection) when configured in this way.

To reproduce this finding, run Pantheon using the `ethereum/core/src/main/resources/dev.json` genesis (which uses the chain ID 2018) and submit a valid, raw transaction with

```
‘params’ : [{‘chainId’ : 2018, …}]
```

This transaction will be rejected by the `MainnetTransactionValidator`.

The severity of this finding is informational because it is unlikely that a production node would be configured in such a way to exercise the bug, since it is the result of using a genesis configuration with no milestones defined.

Exploit Scenario
A valid transaction containing a correct chain ID is rejected by Pantheon, at best resulting in inconsistency with other Ethereum clients, and at worst causing a fork.

Recommendation
In the short term, determine why `MainnetTransactionValidator` does not have knowledge of the chain ID on which Pantheon is running, and fix this bug. In the long term increase test coverage to exercise transactional edge cases.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Arithmetic	Related to arithmetic calculations
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to 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, 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. Code Quality Recommendations

- `util/src/main/java/net/consensys/pantheon/util/uint/UInt256Bytes.java` line 401:

```
static int bitLength(Bytes32 bytes) {
  for (int i = 0; i < SIZE; i++) {
    byte b = bytes.get(i);
    if (b == 0)
      continue;

    return (SIZE * 8) - (i * 8) - (Integer.numberOfLeadingZeros(b & 0xFF) - 3 * 8);
  }
  return 0;
}
```

  Returning at the end of a `for` loop is confusing, and can potentially mask or induce errors in the future. Consider refactoring the code to use different loop semantics.

- Ensure that all serializable classes define a `serialVersionUID`. Currently, only classes derived from `com.google.errorprone.bugpatterns.BugChecker` are missing `serialVersionUID`. Pantheon may never plan to serialize these objects, but the parent class `Bugchecker` implements the Serializable interface. The `serialVersionUID` is declared as a static field within a class that implements the `java.io.Serializable` interface, similarly to the following. A Java IDE can assist in generating UID values.

```
public class DoNotReturnNullOptionals extends BugChecker implements MethodTreeMatcher
{
  private static final long serialVersionUID = 1011858925107209062L;
```

References:

- What is a serialVersionUID and why should I use it?

C. Notes on Cryptographic Libraries

JCE providers

Java supports pluggable JCE providers which allow consumers to pick and choose the underlying implementation for a given set of cryptographic operations. Bouncy Castle is a widely used JCE provider and supports the `secp256k1` ECDSA signatures with RFC 6979 deterministic nonces that are required. In the future, PegaSys may want to investigate explicitly supporting alternate JCE providers for PKCS11 support or cloud providers. For example:

- Azure Key Vault, a Microsoft JCE provider that allows for ECDSA signatures against keys stored securely by Microsoft.
- Sun PKCS#11 provider, a method of bridging PKCS11 APIs (used by hardware security modules) to JCE.

At this time, GCP's KMS and the AWS Cloud HSM v2 do not support `secp256k1` operations; a requirement for considering their use with PegaSys.

Since Java allows end users to specify JCE providers (and preferential ordering) via a `java.security` configuration file if the exact provider is not hard-coded, then the provider chosen can be anything that implements the JCE interfaces. If alternate JCE providers are unsupported then hard-coding is the easy solution. However, if PegaSys decides to support alternate JCE providers that have not been explicitly tested by the PegaSys team, they should run automated health check tests during startup to confirm the JCE provider is performing as expected.

Tink

Google Tink is a multi-language, cross-platform library that provides cryptographic APIs that are secure, easy to use correctly, and harder to misuse. Trail of Bits evaluated Tink as a more secure replacement to Bouncy Castle.

To be an effective replacement, Tink must provide both a quality CSPRNG as well as ECDSA signing using `secp256k1` with nonce generation via RFC 6979. Tink's random class is a very thin wrapper on SecureRandom (which is great), but the ECDSA layer does not expose `secp256k1`, so the library would require maintaining a local patch or convincing Tink to land support for this curve. At this time, Tink is not appropriate for use by PegaSys.

Newer JDK improvements simplify cryptographic operations

Consider accelerating the existing plan to require JDK 10, in order to benefit from its improved cryptographic API support relative to JDK 8. JDK 11 is out now (and JDK 10 is rapidly approaching unsupported, given Oracle's new release strategy).

If you were to require a more recent JDK, you could use its implementations of cryptographic algorithms instead of relying on Bouncy Castle. JDK 10 and later provides its own implementation of the NIST SP 800-90Ar1 `DRBG` recommendations. The JDK 10 (and later) implementation of NIST `DRBG` recommendations would simplify the existing codebase and resolve the issue with the Bouncy Castle DRBG improperly attempting to write to `/dev/urandom` on macOS.

In JDK 10 and later, TLS works by default in OpenJDK. Secure-by-default TLS is not relevant in the current Pantheon codebase (it makes no TLS requests), but if it performs any TLS in a future version, then it would be best to avoid using a runtime like JDK 8 that implements a dangerous-by-default TLS. In a default OpenJDK 8 install, you can't verify a connection without additional work, because OpenJDK 8 builds didn't ship with CA certificates. Only Oracle JRE or Open JDKs from certain distros shipped with CA certificates in JDK 8. JDK 10 and later don't have this issue. JDK 11 also adds the cipher `chacha20poly1305`, which is useful for constrained mobile devices.

And, of course, the usual raft of security hardening and improvement around the JVM itself comes along with each upgrade (JDK 9 Release Notes, JDK 10 Release Notes, JDK 11 Release Notes).

urandom permissions issues when reseeding on macOS

The NIST DRBG implementation in Bouncy Castle attempts to write to the underlying random device on calls to `setSeed`. When passing `-Djava.security.egd=file:/dev/urandom` this causes the code to write to `/dev/urandom`. On Linux this is fine, but macOS does not allow writes to `/dev/urandom`, only `/dev/random`. This additional reseeding is, as discussed in other sections, not necessary. Switch to using a pure `SecureRandom` implementation to resolve this issue, in addition to its other advantages.

D. Additional RLP Unit Tests

The following are additional unit tests we recommend be integrated into Pantheon for better test coverage. They exercise edge cases in RLP length calculations and handling of malformed RLP encodings. Tests `intMaxRLPStringDecode`, `intMaxRLPStringLength`, `intMaxRLPStringInput`, and `decodeIntWithLeadingZeros` fail on the assessed version of Pantheon. See findings TOB-CPP-009, TOB-CPP-010, and TOB-CPP-011 for more information.

```java
package net.consensys.pantheon.ethereum.rlp;

import static org.junit.Assert.assertEquals;

import net.consensys.pantheon.util.bytes.BytesValue;

import org.junit.Test;

import java.util.Random;
import java.util.Stack;

public class RlpUtilsTest {
    private static BytesValue h(String hex) {
        return BytesValue.fromHexString(hex);
    }

    private static String times(String base, int times) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < times; i++)
            sb.append(base);
        return sb.toString();
    }

    private void testLength(byte[] b, Integer expected) {
        if (expected == null) {
            expected = b.length;
        }
        assertEquals(RlpUtils.decodeLength(b, 0), expected.intValue());
    }

    private void testLength(byte[] b) {
        testLength(b, null);
    }
```

```java
private void testLength(BytesValue hex, Integer expected) {
    testLength(hex.extractArray(), expected);
}

private void testLength(BytesValue hex) {
    testLength(hex, null);
}

private void testLength(String hex, Integer expected) {
    testLength(h(hex), expected);
}

private void testLength(String hex) {
    testLength(hex, null);
}

@Test
public void singleByte() {
    testLength("0x01");
}

@Test
public void singleShortElement() {
    testLength("0x81FF");
}

@Test
public void singleBarelyShortElement() {
    testLength("0xb7" + times("2b", 55));
}

@Test
public void singleBarelyLongElement() {
    testLength("0xb838" + times("2b", 56));
}

@Test
public void singleLongElement() {
    testLength("0xb908c1" + times("3c", 2241));
}

@Test
public void assertLongScalar() {
    testLength("0x80");
    testLength("0x01");
```

```
        testLength("0x0F");
        testLength("0x820400");
    }

    @Test(expected = IndexOutOfBoundsException.class)
    public void longScalar_NegativeLong() {
        testLength("0xFFFFFFFFFFFFFFFF");
    }

    @Test
    public void intScalar() {
        testLength("0x80");
        testLength("0x01");
        testLength("0x0F");
        testLength("0x820400");
    }

    @Test
    public void emptyList() {
        testLength("0xc0");
    }

    @Test
    public void simpleShortList() {
        testLength("0xc22c3b");
    }

    @Test
    public void simpleIntBeforeShortList() {
        testLength("0x02");
        testLength("0xc22c3b");
        testLength("0x02c22c3b", 1);
        testLength("0xc22c3b02", 3);
    }

    @Test
    public void simpleNestedList() {
        testLength("0xc52cc203123b");
    }

    @Test
    public void readAsRlp() {
        // Test null value
        testLength("0x80");
```

```java
        testLength("0xc0");
    }

    @Test
    public void raw() {
        testLength("0xc80102c51112c22122");
    }

    @Test
    public void reset() {
        testLength("0xc80102c51112c22122");
    }

    @Test
    public void ignoreListTail() {
        testLength("0xc80102c51112c22122");
    }

    @Test
    public void leaveListEarly() {
        testLength("0xc80102c51112c22122");
    }

    private BytesValueRLPOutput randomRLP(Random random) {
        final BytesValueRLPOutput out = new BytesValueRLPOutput();
        final Stack<Integer> lengths = new Stack<>();
        out.startList();
        lengths.push(0);
        while(!lengths.empty() && (lengths.size() > 1 || random.nextInt(3) > 0)) {
            if (lengths.peek() >= Integer.MAX_VALUE) {
                if (lengths.size() > 1) {
                    out.endList();
                }
                lengths.pop();
                continue;
            }
            switch (random.nextInt(6)) {
                case 0:
                    out.writeByte((byte)random.nextInt(256));
                    lengths.push(lengths.pop() + 1);
                    break;
                case 1:
                    out.writeShort((short)random.nextInt(0xFFFF));
                    lengths.push(lengths.pop() + 2);
                    break;
                case 2:
                    out.writeInt(random.nextInt());
                    lengths.push(lengths.pop() + 4);
                    break;
                case 3:
                    out.writeLong(random.nextLong());
                    lengths.push(lengths.pop() + 8);
                    break;
                case 4:
                    out.startList();
                    lengths.push(0);
                    break;
                case 5:
                    if (lengths.size() > 1) {
                        out.endList();
                        lengths.pop();
                    }
                    break;
            }
        }
        out.endList();
        return out;
    }

    @Test
    public void fuzz() {
        final Random random = new Random();
        for (int i=0; i<1000; ++i) {
            BytesValueRLPOutput out = randomRLP(random);
            assertEquals(RlpUtils.decodeLength(out.encoded().extractArray(), 0),
out.encodedSize());
        }
    }

    @Test
    public void extremelyDeepNestedList() {
        final int MAX_DEPTH = 20000;
        final BytesValueRLPOutput out = new BytesValueRLPOutput();
        int depth = 0;
        for (int i=0; i<MAX_DEPTH; ++i) {
            out.startList();
            depth += 1;
        }
        while (depth > 0) {
            out.endList();
            --depth;
        }
        RlpUtils.decodeLength(out.encoded().extractArray(), 0);
    }

    /*
     * RLP encoded strings, byte arrays, and lists can be up to 256^8 bytes long.
     * This is over twice as big as Long.MAX_VALUE, so confirm that the encoding and
decoding algorithms can handle
     * edge cases with long lengths.
     *
     * The following several tests check for this.
     */

    /**
     * Test how the length calculation handles an incomplete RLP encoding that
reports to be a string of length 2^32.
     * This is larger than Integer.MAX_VALUE, so check that the length calculation
doesn't fail on it due to integer
     * overflow.
     */
    @Test(expected = IndexOutOfBoundsException.class)
    public void intMaxRLPStringLength() {
        RlpUtils.decodeLength(h("0xbc0100000000").extractArray(), 0);
    }

    /**
     * Test how the length calculation handles an incomplete RLP encoding that
reports to be a string of length 2^32.
     * This is larger than Integer.MAX_VALUE, so check that the decoding doesn't fail
on it due to integer overflow.
     */
    @Test
    public void intMaxRLPStringInput() {
        RLP.input(h("0xbc0100000000"));
    }

    @Test
    public void intMaxRLPStringDecode() {
        RLP.decode(BytesValue.wrap(new byte[]{(byte)0xBC, 0x01, 0x00, 0x00, 0x00,
0x00}));
    }

    @Test(expected = MalformedRLPInputException.class)
    public void decodeIntWithLeadingZeros() {
        RLPInput in = RLP.input(h("0x0000D0"));
        RLP.decode(in.raw());
    }
}
```

E. Using the Java SecurityManager

The JVM contains a feature known as the SecurityManager which allows you to restrict your Java application's network, file system, and other core operations. By restricting its privileges, an application can contain the impact of an exploited vulnerability in Java program logic, help preserve system integrity, and reduce the possibility of information disclosures outside of any data that the program is designed to require. Use of the SecurityManager does not mitigate risks of vulnerabilities in the JVM itself or of native code dependencies, but is part of a defense-in-depth strategy.

SecurityManager can be defined both as a policy file (invoked via a command line argument) or programmatically. However, the latter approach allows for potential replacement/removal of the security policy, so it should not be used. Instead, pass

```
        -Djava.security.manager -Djava.security.policy==pegasys.policy
```

where `pegasys.policy` is the policy file that is built for the application.

By default an empty policy file grants no privileges, so a good way to build a restrictive policy would be to start up the application and add a new (minimal) permission for every exception encountered. Alternately, granting full permissions and then ratcheting down on high-risk areas (*e.g.*, file system reads and writes) may be more manageable.

F. Differential Testing with Etheno

Differential Testing, also known as *Differential Fuzzing*, is a technique in which identical inputs are fed to multiple implementations of the same specification in an attempt to detect behavioral differences between the implementations. This approach is ideal for testing Ethereum clients, since the clients must exhibit identical behavior or risk forking the blockchain.

Differential testing of Ethereum clients is challenging because:

1. it requires the clients to be undiscoverable, so other peers do not influence their state;
2. contract addresses and transaction hashes can be different between clients if they have different geneses or have processed different blocks; and
3. there needs to be a way to automatically detect erroneous differences between clients' output.

The JSON RPC multiplexer and testing tool Etheno addresses these challenges.

Using Etheno for Differential Testing

Etheno acts as a JSON RPC client, multiplexing the JSON RPC calls it receives to one or more "real" Ethereum clients, taking care to synchronize contract addresses across the clients. Etheno does this by dynamically rewriting transactions as necessary. It then compares various features such as gas usage and contract creations in order to determine if any of the clients are behaving differently from one another. Discrepancies in behavior causes problems for maintaining consensus between nodes of different clients, and may result in unintended blockchain forks.

First, install Etheno:

```
$ git clone https://github.com/trailofbits/etheno.git
$ cd etheno
$ pip3 install .
```

Alternatively, you can run Etheno in a Docker container:

```
$ docker pull trailofbits/etheno
$ docker run -it trailofbits/etheno
```

Then call Etheno with a list of URLs of Ethereum clients to test:

```
$ etheno http://localhost:8545/ http://localhost:8546/
```

Etheno is also integrated with Geth (and will soon be integrated with Parity). To compare a local Ethereum client to Geth, for example, run:

```
$ etheno --geth http://localhost:8545/
```

You can also provide a genesis file for Geth, e.g., to ensure that it starts with the exact same state as your client:

```
$ etheno --geth --genesis /path/to/genesis.json http://localhost:8545/
```

If your client does not support the eth_sendTransaction call for local accounts, prefix its URL with --raw and Etheno will automatically sign incoming transactions and send them to your client using eth_sendRawTransaction.

```
$ etheno --geth --genesis /path/to/genesis.json \
  --raw http://localhost:8545/
```

Note that use of --raw and --genesis at the same time requires that account private keys be included in the genesis file.

```
"<ACCOUNT_ADDRESS>": {
    "privateKey": "<PRIVATE_KEY>",
    "comment": "private key and this comment are ignored.
                In a real chain, the private key should
                NOT be stored",
    "balance": "9000000000000000000000"
}
```

Etheno performs differential testing automatically, emitting a report at the end of the run.

Automated Fuzzing with Etheno and Echidna

Echidna is a fuzzer/property-based tester of EVM bytecode. It supports sophisticated grammar-based fuzzing campaigns. It is integrated with Etheno and can be used to automatically generate transactions to test against clients. Etheno will automatically prompt you to install Echidna, if necessary. Invoke it by passing `--echidna` to Etheno:

```
$ etheno --geth --genesis /path/to/genesis.json \
  --raw http://localhost:8545/
  --echidna
```

By default, the `--echidna` option deploys a standard fuzzing contract, generates a minimal set of transactions that achieve maximal coverage of the contract, executes those transactions, and exits. There are command-line options to provide a custom contract for Echidna to fuzz.

See below for a sample command to begin a fuzzing campaign against Pantheon with Etheno:

```
$ ./gradlew run -Ppantheon.run.args="--no-discovery
  --datadir=/tmp/pantheontmp --miner-enabled --rpc-enabled
  --miner-coinbase fe3b557e8fb62b89f4916b721be55ceb828dbd73
  --rpc-listen=127.0.0.1:1234 --p2p-listen=127.0.0.1:33333
  --genesis=ethereum/core/src/main/resources/dev.json"
$ etheno --geth --raw http://localhost:1234/
  --genesis ethereum/core/src/main/resources/dev.json
  --echidna
```