



Parity

Security Assessment

Parity Technologies
July 22, 2018

Prepared For:

Raul Romanutti | *Parity Technologies*
raul@parity.io

Prepared By:

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

Jay Little | *Trail of Bits*
jay@trailofbits.com

Andy Ying | *Trail of Bits*
andy@trailofbits.com

Changelog

February 23, 2018: Initial report delivered
June 15, 2018: Added [Appendix D](#) with retest results
July 6, 2018: Added additional retest results

[Executive Summary](#)

[Engagement Goals & Scope](#)

[Coverage](#)

[Project Dashboard](#)

[Rust Recommendations Summary](#)

[Short Term](#)

[Long Term](#)

[Smart Contracts Recommendations Summary](#)

[Short Term](#)

[Long Term](#)

[Rust Findings Summary](#)

[Rust Findings](#)

- [1. Key files may be deleted without authorization during wallet import](#)
- [2. Rust-crypto is not recommended for security-critical usage](#)
- [3. HMAC comparison in do_decrypt is vulnerable to timing attacks](#)
- [4. "single message" crypto operations lack authentication due to using AES-CTR](#)
- [5. Deserialized address field in SafeAccount is not properly sanitized](#)
- [6. Content-Security-Policy is overly permissive](#)
- [7. Confidential information resides in memory for too long](#)
- [8. Parity executables on Windows lack code signatures](#)

[Solidity Findings Summary](#)

[Solidity Findings](#)

- [1. Re-entrancy may lead to stolen ethers](#)
- [2. Missing loop iteration leads to non-removable validator](#)
- [3. Incorrect interface implementation leads to unexpected behavior](#)
- [4. Incorrect conditional prevents fork rejection](#)
- [5. Uninitialized value leads to an unmodifiable owners list](#)
- [6. Race condition may preempt an Ethereum address to email association](#)
- [7. Incorrect interfaces may lead to unexpected behavior](#)
- [8. Incorrect authorization prevents the calling of reporting functions](#)
- [9. "Unrequired" clients can remove a "required" client's privilege](#)
- [10. Missing contract existence check may cause unexpected behavior](#)
- [11. Race condition may lead to content compromise](#)
- [12. Fork re-proposition may prevent owners from accepting or rejecting a fork](#)

- [13. Owners cannot accept or reject re-proposed transactions](#)
- [14. Lack of argument validation may lead to incorrect deletion of badge information](#)
- [15. Deleting clients may lead to incorrect getter values](#)
- [16. Deleting entries may lead to incorrect getter value \(SimpleRegistry\)](#)
- [17. Deleting dapps may lead to incorrect getter value \(DappReg\)](#)
- [18. Deleting badges may lead to incorrect getter value \(BadgeReg\)](#)
- [19. Empty keyServerIp may lead to incorrect keyServersList](#)
- [20. Contracts specify outdated compiler version](#)

[A. Vulnerability Classifications](#)

[B. Code Quality Recommendations](#)

[C. Slither](#)

[D. Fix Log](#)

[Rust Fix Log](#)

[Solidity Fix Log](#)

[Detailed Issue Discussions](#)

[About Trail of Bits](#)

Executive Summary

From January 8 to February 23, Parity Technologies engaged with Trail of Bits to assess targeted components of the Parity wallet and specific smart contracts. The assessed Parity wallet components were written in Rust and JavaScript while the smart contracts were written in Solidity. Trail of Bits conducted this assessment over the course of twelve person-weeks with three engineers.

Trail of Bits completed the assessment using manual, static, and dynamic analysis techniques. The first and second weeks were spent gaining a deep understanding of the [ethstore](#), [ethkey](#), [ethcrypto](#), [rpc](#), and smart contracts codebase and identifying common flaws. In the third and fourth week, we reviewed specific RPC modules, the crypto components, and smart contracts for issues related to data validation, access controls, authentication, cryptography, inheritance, and memory management. During the last two weeks, we continued reviewing smart contracts and Rust modules with additional vigilance concerning data exposure, configuration, and timing issues.

The assessment identified a variety of issues in Parity. The most severe issues may lead to the theft of ethers and the unauthorized deletion of wallets. Other reported issues involved various implications of errors in configuration, cryptography, data validation, and memory management as well as incorrect assumptions regarding timing and contract interfaces.

The Rust code reviewed is of great quality, written with an excellent grasp of idiomatic Rust, and an awareness of security considerations. Code comments are a bit sparse but does not adversely affect the readability of the code. However, the code could make better use of proper primitives for sensitive comparisons in cryptography-related code.

The Solidity code reviewed could use improvement in many areas. The codebase would benefit from unit testing and more detailed documentation. Restructuring contract inheritance and proper use of interfaces would benefit several smart contracts. Effort could be applied to homogenize the codebase by adopting a single coding standard and standardizing on a single Solidity version.

Parity should correct the identified security issues, improve documentation, create unit tests, and consider applying the code quality recommendations. Parity should also consider re-deploying fixed versions of flawed smart contracts.

Trail of Bits has included its Solidity static analyzer, Slither, with the report. Slither detects security flaws in smart contracts, and would have identified multiple flaws described in this report. Using static analysis tools such as Slither to validate code changes may prevent the introduction of future vulnerabilities. [Appendix C](#) provides additional details about Slither.

Engagement Goals & Scope

The goal of the engagement was to evaluate the security of the Parity wallet and smart contracts with the focus on answering the following questions:

- Are the key-generation procedures correct and adherent to modern best practices?
- Do the cryptography components adhere to modern best practices?
- Are the private keys confidential?
- Are account management operations properly authorized and authenticated?
- Are restricted JSON-RPC methods accessible without sufficient permissions?
- Is the ability to intercept communications between core Parity nodes and the UI limited?
- Is HTTP/WebSocket access to nodes only possible through interfaces that are intentionally exposed?
- Is it possible for an attacker to steal ethers from a smart contract?
- Is it possible to block a smart contract in a particular state?
- Do the smart contracts behave as expected? Is the RelaySet smart contract secure?

Scope for the engagement included beta branch of Parity, the `parity-1.8` branch of the `jsonrpc` library, and the `master` branch of the `contracts` repo, specifically the following code:

Functionality Area	Components
Key Generation and Storage	<ul style="list-style-type: none">• crypto module• key generation module• key storage module• js module
RPC	<ul style="list-style-type: none">• jsonrpc-http-server• jsonrpc-ws-server• parity-rpc module (subset that manages private keys)
Smart Contracts ¹	<ul style="list-style-type: none">• KeyServerSet• Operations• Badge contracts• Urlhint• RelaySet• InnerOwnedSet• SimpleRegistrar• SimpleCertifier

¹ Note: the Parity multi-sig wallet smart contract code was not in scope for this engagement.

Coverage

Most of the Rust review was devoted to [ethkey](#), [ethcrypto](#), and [ethstore](#). On the smart contracts side, the primary focus was on [Operations.sol](#) and certifier-related contracts ([ProofOfEmail.sol](#), [SMSVerification.sol](#), and [SimpleCertifier.sol](#)).

Ethkey module. We reviewed the [random](#), [prefix](#), and [brain](#) key generation methods for flaws such as weak entropy and adherence to cryptographic best practices.

Ethcrypto module. We reviewed the [aes](#) and [ECDH/ECIES](#) modules as well as the [key derivation functions](#) for issues related to data exposure and adherence to cryptographic best practices.

Ethstore module. We reviewed the account and vault-management functions for issues related to data exposure, data validation, access controls, and authentication. In addition, we investigated the [cryptographic components](#) of vault and key files. We ensured that the [cryptographic components](#) adhered to best practices.

JS module. We used dynamic analysis to examine the Javascript code for data validation issues in account and vault metadata, with a focus on issues that may lead to cross-site scripting (XSS) vulnerabilities. Due to time constraints, we did not review the JS module for issues related to communication interception between core Parity nodes and the UI.

RPC components. We reviewed the WebSocket RPC authentication mechanism for authentication bypass flaws. Additionally, we examined the RPC function tasked with confirming raw transactions (specifically the RLP decoder). Due to time constraints, we did not finish our investigation into both the WebSocket RPC authentication mechanism and the transaction confirmation process, nor did we review the JSON-RPC [HTTP/WS](#) server.

Smart contracts. We analyzed [BadgeReg.sol](#), [DappReg.sol](#), [GithubHint.sol](#), [key_servers_set.sol](#), [InnerOwnedSet.sol](#), [Operations.sol](#), [ProofOfEmail.sol](#), [RelaySet.sol](#), [SimpleCertifier.sol](#), [SimpleRegistry.sol](#), and [SMSVerification.sol](#). We looked for common Solidity flaws, such as integer overflows, re-entrancy vulnerabilities, and unprotected functions. We also looked for more nuanced flaws, such as logical errors and race conditions. We checked the contracts for errors in memory management and verified proper use of contract inheritance. The high number of issues found in these areas of concern may imply the presence of other vulnerabilities. Due to time constraints, we did not assess the contracts for potential denial of service due to high gas consumption.

Project Dashboard





Application Summary

Name	Parity
Version	364bf48c , 619b1b52 , 7defba32 , cd115965
Type	Wallet, Smart contract
Platform	Rust, Solidity, Javascript








Engagement Summary

Dates	January 8 to February 23, 2018
Method	Whitebox
Consultants Engaged	3
Level of Effort	12 person-weeks

Vulnerability Summary

Total High Severity Issues	3	
Total Medium Severity Issues	13	
Total Low Severity Issues	10	
Total Informational Severity Issues	2	
Total Undetermined Severity Issues	0	
Total	28	

Category Breakdown

Access Controls	1	
Configuration	1	
Cryptography	5	
Data Validation	16	
Patching	1	
Undefined Behavior	2	
Timing	2	
Total	28	

■ dapps ■ SimpleCertifier.sol / SMSVerification.sol ■ Operations.sol
■ InnerOwnedSet.sol / RelaySet.sol ■ GithubHint.sol ■ SimpleRegistry.sol
■ DappReg.sol ■ BadgeReg.sol ■ ethstore ■ ethcrypto ■ ProofOfEmail.sol
■ key_servers_set.sol ■ All smart contracts ■ Parity Windows installation

Rust Recommendations Summary

Short Term

- ❑ **Explicitly whitelist valid sources in the Content Security Policy.** The current policy is too permissive and does not protect the user from some attacks.
- ❑ **Use a constant-time string comparison for sensitive string comparisons in `Crypto::do_decrypt`.** Non-constant-time comparisons expose the code to timing attacks.
- ❑ **Move to AES-GCM or another authenticated cipher mode in the ECIES module.** AES-CTR is malleable, so attackers could modify messages in some circumstances.
- ❑ **Generate a new UUID for imported accounts and verify that there are no conflicts during account import.** Identical UUIDs encountered during wallet imports may lead to wallet deletion.
- ❑ **Do not trust the value of address when loading `KeyFiles` from the root vault.** Consider protecting the root vault with a password by default, and warning the user about the consequences of an unprotected root vault.
- ❑ **Implement the `Drop` trait for objects holding confidential content.** An information leak may exist if sensitive contents in memory are not cleared.
- ❑ **Ensure that all binaries included in the Parity installation are digitally signed.** The lack of signing may allow an attacker to tamper with the binary and hinders endpoint security efforts.

Long Term

❑ **Sign all Windows executables with SHA1 and SHA2.** Signing with only SHA1 presents a security risk. Using multiple signatures prevents length-extension attacks. Signing with SHA1 and SHA2 is required by Windows 7 and later to treat an executable as signed.

❑ **Develop guidelines and policies to better identify and handle untrusted input.** Assume that all inputs may be malicious and sanitize them.

❑ **Eliminate the cryptographic dependency on unreviewed crypto libraries.** Parity depends on [rust-crypto](#) and [libsecp256k1](#). Neither [rust-crypto](#) or [libsecp256k1](#) have been highly reviewed, formally proven, or thoroughly researched. Consider conducting a detailed security assessment of these libraries or migrating to more highly reviewed alternatives.

Smart Contracts Recommendations Summary

Short Term

- ❑ **Change the loop iteration condition in the OwnedSet constructor.** The constructor loop does not include the last initial validator.
- ❑ **Rename get to getData in SimpleCertifier.sol and SMSVerification.sol and define Certifier as an interface.** Inconsistencies in the contract interfaces may lead to unexpected behavior for third-party applications or contracts.
- ❑ **Prevent forks from being re-proposed in Operations.** A fork re-proposition will prevent some owners from voting on a fork.
- ❑ **Prevent getters from accessing deleted objects.** Operations, SimpleRegistry, DappReg and BadgeReg getters can return data from deleted objects. This may lead to unexpected behavior for a third-party.
- ❑ **Change the condition in the Operations.only_unratified modifier.** In its current state, the modifier prevents a fork from being rejected.
- ❑ **Initialize outerSet in InnerOwnedSet.** outerSet is uninitialized and cannot be changed in InnerOwnedSet. This breaks InnerOwnedSet features.
- ❑ **Make GithubHint.hint and GithubHint.hintURL return a boolean denoting the function's success and throw an error in case of error.** The user should be alerted if these functions fail. This would mitigate a race condition attack.
- ❑ **Use the sender address from the ProofOfEmail request for the confirmation process.** A race condition may allow an attacker to fraudulently change the email address associated with the sender ethereum address; additional checks are therefore needed.
- ❑ **Remove ReverseRegistry and Certifier from ProofOfEmail.sol and import the contracts from Registry.sol and Certifier.sol.** The current state of the contracts is inconsistent with their original versions.
- ❑ **Prevent an Operations's client from transferring its name to a client address already associated with a name.** An unauthorized client is able to drop the authorization of another client by transferring its name.

- ❑ **Prevent transactions from being re-proposed.** A transaction re-proposition will prevent some owners from voting on a transaction.
- ❑ **Ensure that `KeyServerSet.addKeyServer` is not applied to an empty-string IP.** Empty-string IPs will create invalid entries in the `KeyServerSet`.
- ❑ **Remove the inline assembly code in `OuterSet`.** The inline assembly code lacks security checks that are provided by Solidity primitives.
- ❑ **In `Operation.checkProxy`, delete the transaction from proxy before its execution.** Executing the transaction before its deletion may lead to a re-entrancy attack.
- ❑ **Ensure that badge registration and unregistration only happens on valid names and addresses.** The current lack of parameter validation in `BadgeReg` may lead to a corrupt badges list.
- ❑ **Ensure that all code can be compiled without warnings with the latest version of the Solidity compiler.** This will ensure that any new checks are utilized and all warnings are surfaced appropriately.

Long Term

- ❑ **Avoid state changes after an external call.** Apply the [check-effects-interactions](#) pattern to prevent re-entrancy vulnerabilities.
- ❑ **Carefully review the [Solidity documentation](#).** In particular, any section that contains a warning must be carefully understood since it may lead to unexpected or unintentional behavior.
- ❑ **Check all function parameters for all unexpected values.** The lack of checks on function parameters may lead to broken function behavior.
- ❑ **Consider that all the information is public before being accepted on the blockchain.** Data sent to Ethereum is public and can be read and used by anyone -- even before being executed by the blockchain.
- ❑ **Create unit tests for all Solidity code.** Ensure that unit tests properly cover all the features of the contract. High unit test coverage enables earlier detection of vulnerabilities and identifies issues added through code modifications.
- ❑ **Do not copy and paste code; import instead.** Several issues were introduced by copying code from another Solidity file.
- ❑ **Avoid inline assembly code where possible.** Writing assembly code is complex and may introduce vulnerabilities.
- ❑ **Document the expected behavior of smart contracts.** Write documentation for each contract, including its expected behavior. Proper documentation aids code reviews and helps identify deviations from expected behavior.
- ❑ **Ensure that all variables are initialized.** Uninitialized variables will lead to unexpected behavior and may introduce vulnerabilities.
- ❑ **Ensure that all variables are used.** Unused variables should be removed since they add unnecessary complexity and cost for the contract.

Rust Findings Summary

#	Title	Type	Severity
1	Key files may be deleted without authorization during wallet import	Data Validation	High
2	Rust-crypto is not recommended for security-critical usage	Cryptography	High
3	HMAC comparison in do_decrypt is vulnerable to timing attacks	Cryptography	Medium
4	"single message" crypto operations lack authentication due to using AES-CTR	Cryptography	Medium
5	Deserialized address field in SafeAccount is not properly sanitized	Data Validation	Medium
6	Content-Security-Policy is overly permissive	Configuration	Low
7	Confidential information resides in memory for too long	Cryptography	Low
8	Parity executables on Windows lack code signatures	Cryptography	Low

Rust Findings

1. Key files may be deleted without authorization during wallet import

Severity: High

Difficulty: High

Type: Data Validation

Finding ID: TOB-Parity-014

Target: ethstore/src/dir/disk.rs, ethstore/src/ethstore.rs

Description

Importing two wallets with the same account ID in rapid succession (within a second) will result in the unauthorized deletion of the earlier-imported wallet.

When a wallet is imported into Parity, it is stored on the filesystem with a dynamically generated name. The filename is generated in `DiskDirectory::insert` ([disk.rs#L205](#)) and consists of a second-granularity timestamp and an account ID.

```
fn insert(&self, account: SafeAccount) -> Result<SafeAccount, Error> {
    // build file path
    let filename = account.filename.as_ref().cloned().unwrap_or_else(|| {
        let timestamp = time::strftime("%Y-%m-%dT%H-%M-%S",
            &time::now_utc()).expect("Time-format string is valid.");
        format!("UTC--{}Z--{}", timestamp, Uuid::from(account.id))
    });

    self.insert_with_filename(account, filename)
}
```

Figure 1: The insert implementation

The account ID is a UUID that is taken directly from a JSON file uploaded by the user ([ethstore.rs#L161](#)). There are no checks to verify that the uploaded account ID does not conflict with the existing account IDs.

```
fn import_wallet(&self, vault: SecretVaultRef, json: &[u8], password: &str)
-> Result<StoreAccountRef, Error> {
    let json_keyfile = json::KeyFile::load(json).map_err(|_|
        Error::InvalidKeyFile("Invalid JSON format".to_owned()))?;
    let mut safe_account = SafeAccount::from_file(json_keyfile, None);
    let secret = safe_account.crypto.secret(password).map_err(|_|
        Error::InvalidPassword)?;
    safe_account.address = KeyPair::from_secret(secret)?.address();
}
```

```
self.store.import(vault, safe_account)
}
```

Figure 2: The import_wallet implementation

If two wallets with the same account ID are imported during the same second interval to the same vault, they will share the same file path. Sharing the same file path results in the earlier-imported wallet being overwritten by the later-imported wallet.

Exploit Scenario

Bob creates a wallet. Carl imports a wallet some time later with Bob's account ID. The server storing the wallets encounters a failure that requires all the accounts to be restored. The automated restoration process imports Bob's wallet and then Carl's wallet to the same vault within the same second. Bob's wallet key file is subsequently overwritten by Carl's wallet.

Recommendation

Generate a new UUID for imported accounts.

Verify that there are no conflicting account IDs during import.

Long term, consider creating unit tests that cover a wider range of application functionality.

2. Rust-crypto is not recommended for security-critical usage

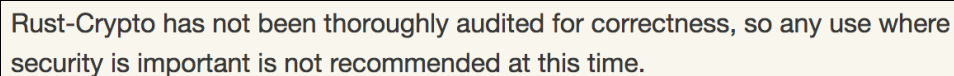
Severity: High
Type: Cryptography
Target: ethcrypto

Difficulty: High
Finding ID: TOB-Parity-028

Description

The rust-crypto crate has never received a thorough security assessment and should not be considered for security-critical usage. Many of the Parity encryption algorithms, encryption modes, and hashing algorithms are based on the rust-crypto crate.

The [rust-crypto crate homepage](#) explicitly mentions that it is not recommended for applications where security is important.

A screenshot of a warning message from the rust-crypto crate homepage. The text is enclosed in a light yellow box with a thin black border. The text reads: "Rust-Crypto has not been thoroughly audited for correctness, so any use where security is important is not recommended at this time."

Rust-Crypto has not been thoroughly audited for correctness, so any use where security is important is not recommended at this time.

Figure 1: Screenshot from rust-crypto crates homepage

Cryptography libraries are notoriously difficult to implement correctly and can fail in catastrophic ways due to seemingly minor flaws.

Exploit Scenario

A critical security issue is discovered one of rust-crypto's cryptographic algorithms which Parity uses. Alice, a malicious actor, exploits this issue to decrypt sensitive Parity key files.

Recommendation

Short term, minimize the use of the rust-crypto crate and prepare to migrate away from it.

Long term, use a more conservative, better reviewed library for cryptographic operations. Consider using a Rust wrapper to OpenSSL or the Rust [Ring](#) library.

References

- [CVE-2016-7798](#): Improper GCM initialization vector setup by the OpenSSL Ruby gem
- [CVE-2014-1266](#): Apple's SSL/TLS bug
- [The many, many ways that cryptographic software can fail](#)

3. HMAC comparison in do_decrypt is vulnerable to timing attacks

Severity: Medium

Difficulty: Medium

Type: Cryptography

Finding ID: TOB-Parity-012

Target: ethstore/src/account/crypto.rs

Description

Crypto::do_decrypt ([crypto.rs#L142](#)) uses a non-constant time string comparison to verify HMAC hash equality.

```
let mac = crypto::derive_mac(&derived_right_bits,
&self.ciphertext).keccak256();

if mac != self.mac {
    return Err(Error::InvalidPassword);
}

let mut plain: SmallVec<u8; 32> = SmallVec::new();
```

Figure 1: The do_decrypt implementation

This method of comparison exposes a timing side channel that allows malicious individuals to determine the correct HMAC hash for arbitrary content.

Exploit Scenario

Bob alters the ciphertext of a vault key file with specially crafted content that modifies important metadata. He repeatedly makes strategic modifications to each byte in the HMAC value of the ciphertext in the vault key file while triggering the vault key file decryption operation. A statistical analysis of the elapsed operation times will reveal a difference between the correct byte in the HMAC and the wrong byte in the HMAC. Eventually, Bob crafts a valid HMAC for his payload content.

Recommendation

Use a constant-time string comparison such as `subtle::arrays_equal` instead of `==` or `!=` for sensitive string comparisons.

References

- [Constant-time Toolkit](#): Reference library of constant-time implementations
- [Why Constant-Time Crypto?](#)
- [Compare secret strings in constant time](#)

4. "single message" crypto operations lack authentication due to using AES-CTR

Severity: Medium

Type: Cryptography

Target: ethcrypto/src/lib.rs

Difficulty: Medium

Finding ID: TOB-Parity-013

Description

In the ECIES module, `encrypt_single_message` and `decrypt_single_message` use AES-CTR. CTR mode is malleable, so under some circumstances attackers could modify encrypted messages without access to the keys. This could allow attackers to effectively impersonate some party communicating using these facilities.

`decrypt_single_message` is used by the SecretStore server. We did not further investigate the SecretStore server because it is not in-scope for the review. However, a cursory review identified that `decrypt_single_message` decrypts ciphertext received over the network.

Exploit Scenario

Alice, a malicious user, uses Wireshark to capture network traffic meant for SecretStore. She extracts a message and strategically tampers with it to alter certain critical fields. Alice sends the crafted message to the SecretStore to gain access to unauthorized functionality.

Recommendation

Use AES-GCM, an authenticated encryption mode. Using an AEAD construction like AES-GCM will provide a guarantee of integrity, rather than just confidentiality.

Refer to [Thomas Ptacek's Cryptographic Right Answers](#) whenever needed for new code.

References

- [Counter Mode Security: Analysis and Recommendations](#) (see Section 2.1)
- [Evaluation of Some Blockcipher Modes of Operation](#)

5. Deserialized address field in SafeAccount is not properly sanitized

Severity: Medium

Difficulty: High

Type: Data Validation

Finding ID: TOB-Parity-021

Target: ethstore/src/dir/disk.rs

Description

The address field in SafeAccount is improperly sanitized after being deserialized from a KeyFile in the root vault. This may allow malicious actors to trick unsuspecting users into fraudulent transactions.

Wallets are stored by default as KeyFile ([key_file.rs#L45](#)) structures serialized into JSON in the root vault.

```
#[derive(Debug, PartialEq, Serialize)]
pub struct KeyFile {
    pub id: Uuid,
    pub version: Version,
    pub crypto: Crypto,
    pub address: H160,
    pub name: Option,
    pub meta: Option,
}
```

Figure 1: The KeyFile structure implementation

EthMultiStore::reload_accounts ([ethstore.rs#L296](#)) is responsible for loading KeyFile objects and adding them, by address, to the accounts reference cache.

```
fn reload_accounts(&self) -> Result<(), Error> {
    let mut cache = self.cache.write();

    let mut new_accounts = BTreeMap::new();
    for account in self.dir.load()? {
        let account_ref = StoreAccountRef::root(account.address);
        new_accounts
            .entry(account_ref)
            .or_insert_with(Vec::new)
            .push(account);
    }
    for (vault_name, vault) in &*self.vaults.lock() {
        for account in vault.load()? {
            let account_ref =
StoreAccountRef::vault(vault_name, account.address);
```

```

        new_accounts
            .entry(account_ref)
            .or_insert_with(Vec::new)
            .push(account);
    }
}

mem::replace(&mut *cache, new_accounts);
Ok(())
}

```

Figure 2: The reload_accounts implementation

During the process of loading KeyFile, the address field is never sanitized ([disk.rs#L272](#)).

```

fn read<T>(&self, filename: Option<String>, reader: T) ->
Result<SafeAccount, Error> where T: io::Read {
    let key_file = json::KeyFile::load(reader).map_err(|e|
Error::Custom(format!("{:?}", e)))?;
    Ok(SafeAccount::from_file(key_file, filename))
}

```

Figure 3: The read implementation

It is possible for a malicious actor with filesystem access to modify the actor's address such that the displayed address differs from the actual address as derived from the KeyFile's secret.

Exploit Scenario

David is planning to send Carl some ether. Bob modifies Carl's serialized JSON KeyFile on the filesystem so that Carl's address is changed to one that Bob controls. Bob forces the Parity server to restart. David does not notice Carl's changed address in the Parity UI and sends the ether to an address he believes Carl controls. The transaction goes through. Carl gets notified that he received the funds. However, the funds are actually sent to Bob.

Recommendation

Do not trust the value of address when loading KeyFiles from the root vault. Consider making the root vault password-protected by default and warning the user about the consequences of an unprotected root vault.

6. Content-Security-Policy is overly permissive

Severity: Low

Type: Configuration

Target: dapps/src/handlers/mod.rs

Difficulty: High

Finding ID: TOB-Parity-001

Description

The Content Security Policy for the Parity UI (accessible at <http://127.0.0.1:8180>) is overly permissive. In the presence of an XSS vulnerability, the applied Content Security Policy may be insufficient to block injected malicious Javascript code.

The `script-src` directive specifies valid Javascript sources. `script-src` is set to `'self'` `'unsafe-inline'` `'unsafe-eval'` which allows for the use of inline resources, the `eval` JavaScript function, and script content from the origin of the served document.

The `object-src` directive specifies valid sources for object, embed, and applet elements. `object-src` is unset. It will default to the value specified by `default-src`. In this case, that value is `self` and indicates that object, embed, and applet elements served from the origin of the served document are valid sources.

Evaluated CSP as seen by a browser supporting CSP Version 3			expand/collapse all
✓	connect-src		▼
✓	frame-src		▼
✓	child-src		▼
✓	img-src		▼
✓	style-src		▼
✓	font-src		▼
❗	script-src		▲
?	'self'	'self' can be problematic if you host JSONP, Angular or user uploaded files.	
❗	'unsafe-inline'	'unsafe-inline' allows the execution of unsafe in-page scripts and event handlers.	
?	'unsafe-eval'	'unsafe-eval' allows the execution of code injected into DOM APIs such as eval().	
✓	worker-src		▼
✓	default-src		▼
✓	sandbox		▼
✓	form-action		▼
✓	block-all-mixed-content		▼
✓	frame-ancestors		▼
❗	object-src [missing]	Can you restrict object-src to 'none'?	▼

Figure 1: Easily evaluate potential CSP configurations with Google's [CSP Evaluator](#)

Exploit Scenario

In conjunction with an XSS vulnerability, an attacker may leverage the overly permissive Content Security Policy to easily evaluate or inject arbitrary Javascript code into the Parity UI. The attacker may be able to steal account passwords and perform other malicious actions via hooking RPC handling functions with malicious code running in the Parity UI.

Recommendation

Explicitly whitelist valid sources in the Content Security Policy.

- Consider setting `object-src` to `'none'` if no embedded content or browser extensions are loaded via `<object>` or `<embed>` tags.
- Consider using nonces on inline Javascript to whitelist these code blocks with CSP.
- Consider whitelisting specific domains and URLs for `worker-src` and `style-src`.

References

- [Google Reference on Content Security Policy](#)
- [Google Content Security Policy Evaluator](#)
- [MDN: Content-Security-Policy script-src](#)
- [MDN: Content-Security-Policy object-src](#)

7. Confidential information resides in memory for too long

Severity: Low

Type: Cryptography

Target: ethstore/src/ethstore.rs

Difficulty: Undetermined

Finding ID: TOB-Parity-022

Description

Confidential information such as keys and passwords are not zeroed immediately after use. For instance, the `secret` variable below contains the unencrypted key for the account to be imported ([ethstore.rs#L164](#)).

```
fn import_wallet(&self, vault: SecretVaultRef, json: &[u8], password:
&str) -> Result {
    let json_keyfile = json::KeyFile::load(json).map_err(|_|
Error::InvalidKeyFile("Invalid JSON format".to_owned()))?;
    let mut safe_account = SafeAccount::from_file(json_keyfile,
None);
    let secret = safe_account.crypto.secret(password).map_err(|_|
Error::InvalidPassword)?;
    safe_account.address = KeyPair::from_secret(secret)?.address();
    self.store.import(vault, safe_account)
}
```

Figure 1: The `import_wallet` implementation

After `import_wallet` completes, `secret` becomes out of scope and is deallocated. Despite the deallocation, the confidential contents of the key still reside at the memory location once occupied by `secret`.

Exploit Scenario

Bob's computer is compromised by Alice. Alice scrapes memory to instantly retrieve the key to Bob's wallet. She does not need to wait for Bob to use or expose the key because it is already in memory. Bob has no time to detect that his computer is compromised before Alice succeeds in stealing his wallet.

Bob runs Parity and experiences a crash. Bob generates a memory dump and posts it to internet forum in search of a resolution. Unknown to Bob, the memory dump contains his secret key. Alice carves the secret key from the memory dump and steals Bob's wallet.

Recommendation

Implement the `Drop` trait for objects holding confidential content. The drop function should overwrite the confidential content with zeroes.

References

- [Should I delete cryptographic data from memory?](#)

8. Parity executables on Windows lack code signatures

Severity: Low

Difficulty: Low

Type: Cryptography

Finding ID: TOB-Parity-027

Target: Parity Windows installation

Description

Several Parity executables on Windows (ethkey.exe, ethstore.exe, parity-evm.exe, and uninstall.exe) are not digitally signed.

```
C:\Program Files\Parity Technologies\Parity\ethkey.exe:
    Verified:    Unsigned
    Link date:   10:35 AM 2/19/2018
-- snipped --
C:\Program Files\Parity Technologies\Parity\ethstore.exe:
    Verified:    Unsigned
    Link date:   10:35 AM 2/19/2018
-- snipped --
C:\Program Files\Parity Technologies\Parity\parity-evm.exe:
    Verified:    Unsigned
    Link date:   10:35 AM 2/19/2018
-- snipped --
C:\Program Files\Parity Technologies\Parity\parity.exe:
    Verified:    Signed
    Signing date: 10:36 AM 2/19/2018
    Publisher:   ETH CORE LIMITED
-- snipped --
C:\Program Files\Parity Technologies\Parity\ptray.exe:
    Verified:    Signed
    Signing date: 10:36 AM 2/19/2018
    Publisher:   ETH CORE LIMITED
-- snipped --
C:\Program Files\Parity Technologies\Parity\uninstall.exe:
    Verified:    Unsigned
    Link date:   7:55 PM 7/24/2016
-- snipped --
```

Figure 1: [sigcheck.exe](#) output on the contents of Parity's installation folder on Windows

Exploit Scenario

Bob downloads the Parity wallet software from the internet and wants to verify that the copy received is legitimate. He cannot verify their integrity by reviewing code signatures.

The enterprise security team at Widgets Inc. regularly reviews the installed and running software on corporate workstations. They cannot easily validate that Parity software is not backdoored or illegitimate since it lacks code signatures.

Recommendation

Ensure that all executables included in the Parity installation are digitally signed. Follow current industry standards by signing the Windows executables with SHA1 and SHA2.

Create a pre-release checklist for all new releases of the Parity wallet. On the checklist, ensure that all executables on every operating system are code signed prior to release.

Solidity Findings Summary

#	Title	Type	Severity
1	Re-entrancy may lead to stolen ethers	Data Validation	High
2	Missing loop iteration leads to non-removable validator	Data Validation	Medium
3	Incorrect interface implementation leads to unexpected behavior	Undefined Behavior	Medium
4	Incorrect conditional prevents fork rejection	Data Validation	Medium
5	Uninitialized value leads to an unmodifiable owners list	Data Validation	Medium
6	Race condition may preempt an Ethereum address to email association	Timing	Medium
7	Incorrect interfaces may lead to unexpected behavior	Undefined Behavior	Medium
8	Incorrect authorization prevents the calling of reporting functions	Access Controls	Medium
9	"Unrequired" clients can remove a "required" client's privilege	Data Validation	Medium
10	Missing contract existence check may cause unexpected behavior	Data Validation	Medium
11	Race condition may lead to content compromise	Timing	Medium
12	Fork re-proposition may prevent owners from accepting or rejecting a fork	Data Validation	Low
13	Owners cannot accept or reject re-proposed transactions	Data Validation	Low
14	Lack of argument validation may lead to incorrect deletion of badge information	Data Validation	Low
15	Deleting clients may lead to incorrect getter values	Data Validation	Low

16	Deleting entries may lead to incorrect getter value (SimpleRegistry)	Data Validation	Low
17	Deleting dapps may lead to incorrect getter value (DappReg)	Data Validation	Low
18	Deleting badges may lead to incorrect getter value (BadgeReg)	Data Validation	Low
19	Empty keyServerIp may lead to incorrect keyServersList	Data Validation	Informational
20	Contracts specify outdated compiler version	Patching	Informational

Solidity Findings

1. Re-entrancy may lead to stolen ethers

Severity: High

Type: Data Validation

Target: Operations.sol

Difficulty: High

Finding ID: TOB-Parity-024

Description

The Operations contract allows authorized users to send ethers. A re-entrancy vulnerability may allow a malicious user to send more ethers than expected.

Once a transaction is confirmed, the checkProxy function is invoked ([Operations.sol#L278-L284](#)):

```
function checkProxy(bytes32 _txid) internal
when_proxy_confirmed(_txid) returns (uint txSuccess) {
    var tx = proxy[_txid];
    var success = tx.to.call.value(tx.value).gas(tx.gas)(tx.data);
    TransactionRelayed(_txid, success);
    txSuccess = success ? 2 : 1;
    delete proxy[_txid];
}
```

Figure 1: The checkProxy implementation

The transaction is executed before it is deleted from the proxy list. A malicious transaction destination may be able to confirm the transaction a second time prior to the transaction's deletion. As a result, the ethers will be sent multiple times.

To exploit this vulnerability, the attacker needs to have access to an authorized client that is not needed to confirm the transaction. It is expected that all the authorized clients are needed to confirm a transaction. However, this assumption can be broken, for example, by incorrectly initializing a contract.

Exploit Scenario

The number of authorized clients to confirm a transaction is set to two, but three authorized clients are added during initialization. Bob's smart contract is one of the authorized clients. There is a pending transaction with one ether to Bob's smart contract. The transaction is executed and Bob's smart contract fallback function is triggered. The

fallback function confirms the transaction a second time. As a result, the transaction is executed twice and two ethers are sent instead of one.

Recommendation

Delete the transaction from proxy prior to execution.

Avoid state changes after an external call. Apply the [check-effects-interactions](#) pattern.

2. Missing loop iteration leads to non-removable validator

Severity: Medium

Difficulty: Low

Type: Data Validation

Finding ID: TOB-Parity-002

Target: InnerOwnedSet.sol

Description

An improper loop iteration in the constructor functions prevents the removal of a consensus validator from the OwnedSet and InnerOwnedSet contracts.

The OwnedSet and InnerOwnedSet contracts are validator sets used in consensus. Both of their constructors take one argument: a list of initial validator addresses called `_initial`. This list of addresses is used in three fields:

1. `pending` (copy of `_initial`)
2. `validators` (copy of `_initial`)
3. `pendingStatus`

In the constructors, the for loop used to initialize a `pendingStatus` `AddressStatus` object has an erroneous end condition. A correct iteration would result in the for loop terminating after handling the address at index `_initial.length-1`. However, the end condition is when the iteration index is greater than or equal to `_initial.length-1`, thus terminating the loop after handling the address at index `_initial.length-2`. This causes the end condition to be off by one, thus skipping initialization of the last item in `_initial`.

```
function OwnedSet(address[] _initial) public {
    pending = _initial;
    for (uint i = 0; i < _initial.length - 1; i++) {
        pendingStatus[_initial[i]].isIn = true;
        pendingStatus[_initial[i]].index = i;
    }
    validators = pending;
}
```

Figure 1: The OwnedSet constructor function

There are two side effects to this state:

1. The function `reportBenign()` cannot be called on the last address
2. The last address cannot be removed as a validator via `removeValidator()`

Below is an example state of an OwnedSet after creation:

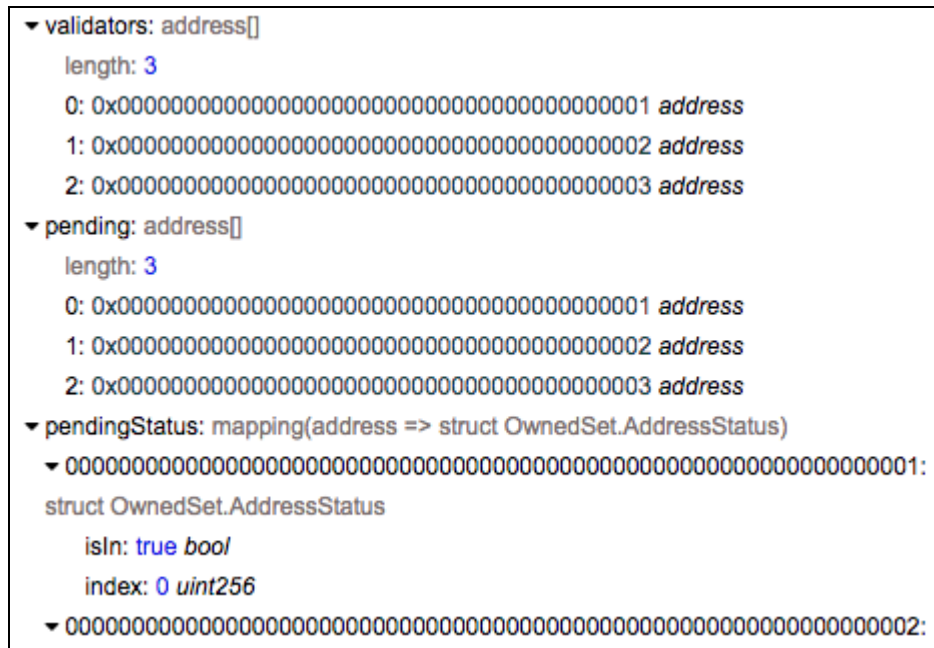


Figure 2: State of an OwnedSet after creation with three addresses

Exploit Scenario

Alice creates an OwnedSet contract with Bob and Eve as consensus validators. Alice no longer trusts Eve and attempts to remove Eve as a validator via `removeValidator()`. This call will fail and Eve will remain in the trusted validators array.

Recommendation

Remove the subtract-by-one in the end condition of the for loop to resolve the coding flaw.

Consider improving unit test coverage. This vulnerability could have been discovered through unit tests that cover OwnedSet and InnerOwnedSet contract operations.

Review each contract creation transaction for OwnedSet and InnerOwnedSet to determine the last element of the array passed to the constructor.

References

<https://paritytech.github.io/wiki/Validator-Set>

3. Incorrect interface implementation leads to unexpected behavior

Severity: Medium

Difficulty: Low

Type: Undefined Behavior

Finding ID: TOB-Parity-003

Target: SimpleCertifier.sol and SMSVerification.sol

Description

SimpleCertifier and ProofOfSMS inherit from Certifier but do not implement the getData function. As a result, any use of this function leads to an incorrect result.

Certifier implements the function getData ([Certifier.sol#L11](#)):

```
function getData(address _who, string _field) constant public returns
(bytes32) {}
```

Figure 1: The Certifier.getData implementation

In the Certifier implementation, getData always returns 0.

SimpleCertifier and ProofOfSMS inherit from Certifier, but this function is not overridden. As a result, any calls to it will return 0. This may lead to unexpected behavior in third-party applications or contracts.

An example of misuse is present in [MasterCertifier.sol#L37-L42](#)

```
function getData(address _who, string _field) public constant returns
(bytes32) {
    for (uint i = 0; i < certifiers.length; ++i) {
        if (certifiers[i].certified(_who)) {
            return certifiers[i].getData(_who, _field);
        }
    }
}
```

Figure 2: The MasterCertifier.getData implementation

On a related note, the function get(address _who, string _field) ([SimpleCertifier.sol#L28](#), [SMSVerification.sol#L31](#)) has the expected behavior of getData. We believe this function was incorrectly named, and should actually be named getData.

Exploit Scenario

Bob creates a smart contract that uses ProofOfSMS. His smart contract performs some checks on the certifier's data by calling getData. Since getData always returns 0, Bob's smart contract does not work.

Recommendation

Rename get to getData in [SimpleCertifier.sol#L28](#) and [SMSVerification.sol#L31](#).

Define Certifier ([Certifier.sol#L7-L14](#)) as an [interface](#) instead of an abstract contract. getData ([Certifier.sol#L11](#)), getAddress ([Certifier.sol#L12](#)), and getUint ([Certifier.sol#L13](#)) should not be implemented in Certifier. This would have prevented the introduction of the issue.

Consider carefully reviewing the interface of a contract. If a contract is meant to be only composed of headers, use [interface](#) instead of [abstract contract](#).

Note: We found [OprahBadge.sol](#) was also vulnerable to this issue (out of scope).

4. Incorrect conditional prevents fork rejection

Severity: Medium

Type: Data Validation

Target: Operations.sol

Difficulty: Low

Finding ID: TOB-Parity-006

Description

An incorrect conditional in the `only_unratified` modifier prevents forks from being rejected. In the `Operations` contract, only a single fork can be proposed at a time. An existing fork proposal needs to be accepted prior to creating new fork proposals.

`rejectFork` is called to reject a fork ([Operations.sol#162-167](#)):

```
function rejectFork() only_when_proposed only_undecided_client_owner
only_unratified {
    var newClient = clientOwner[msg.sender];
    fork[proposedFork].status[newClient] = Status.Rejected;
    ForkRejectedBy(newClient, proposedFork);
    noteRejected(newClient);
}
```

Figure 1: The `rejectFork` implementation

`rejectFork` has the `only_unratified` modifier ([Operations.sol#L291-L292](#)):

```
modifier only_ratified{ if (!fork[proposedFork].ratified) throw; _; }
modifier only_unratified { if (!fork[proposedFork].ratified) throw; _; }
```

Figure 2: The `only_ratified` and `only_unratified` modifiers

`only_unratified` and `only_ratified` contain the same code. They both ensure that the fork has been ratified. Since a fork is only ratified when accepted, it is not possible to reject a fork.

Once a fork is under proposal, no other forks can be proposed. The current fork has to be accepted in order to propose new forks.

Exploit Scenario

Bob proposes a fork. He realizes the fork proposal was a mistake and wants to reject the fork. He attempts to reject the fork but the rejection invocation fails. Bob has to accept the erroneous proposed fork in order to propose the amended fork.

Recommendation

Fix the `only_unratified` modifier.

Create unit tests. A test on the [rejection API](#) could have found this vulnerability.

5. Uninitialized value leads to an unmodifiable owners list

Severity: Medium

Difficulty: Low

Type: Data Validation

Finding ID: TOB-Parity-007

Target: InnerOwnedSet.sol and RelaySet.sol

Description

An uninitialized OuterSet in InnerSet renders the owners list in InnerOwnedSet immutable.

InnerOwnedSet inherits from InnerSet ([InnerOwnedSet.sol#L22](#)):

```
contract InnerOwnedSet is Owned, InnerSet {
```

Figure 1: The InnerOwnedSet contract

InnerSet has an [outerSet variable](#):

```
contract InnerSet {  
    OuterSet public outerSet;
```

Figure 2: The InnerSet contract

Since outerSet is never initialized in InnerSet and InnerOwnedSet, outerSet has an immutable value of 0.

outerSet is used in the only_outer modifier ([RelaySet.sol#L82-L85](#)):

```
modifier only_outer() {  
    require(msg.sender == address(outerSet));  
    _;  
}
```

Figure 3: The only_router modifier

... which is used by the finalizeChange function ([InnerOwnedSet.sol#L84](#)):

```
function finalizeChange() public only_outer {
```

Figure 4: The finalizeChange function

With an uninitialized outerSet, only_outer will always cause the finalizeChange function to fail.

Similarly, `initiateChange` attempts to call a method of `outerSet` ([InnerOwnedSet.sol#L79-L80](#)).

```
function initiateChange() private {
    outerSet.initiateChange(block.blockhash(block.number - 1),
    getPending());
}
```

Figure 5; The `initiateChange` function

0 is a [valid address](#) that does not contain code. A high-level call to an address [without code fails](#):

Function calls cause exceptions if the called contract does not exist (in the sense that the account does not contain code) or if the called contract itself throws an exception or goes out of gas.

As a result, the call to `outerSet` fails. Since `addValidator` and `removeValidator` both call `initiateChange`, this failure makes adding and removing validators impossible.

Exploit Scenario

Alice, Bob and Eve are the three initial validators of `InnerOwnedSet`. Alice and Bob realize that Eve is malicious and want to remove her from the validators list, but are not able to do so.

Recommendation

Initialize `outerSet`.

Ensure that all variables are correctly initialized.

Create unit tests for this functionality. A test on the owners modification functionality may have found this vulnerability.

6. Race condition may preempt an Ethereum address to email association

Severity: Medium

Difficulty: High

Type: Timing

Finding ID: TOB-Parity-015

Target: ProofOfEmail.sol

Description

A race condition in ProofOfEmail may allow an attacker to preempt an Ethereum-address-to-email association.

ProofOfEmail associates a user Ethereum address with an email.

When the user requests email verification, the [server associates the SHA3 hash](#) of code to an email hash in the puzzles mapping ([ProofOfEmail.sol#L62-L65](#)):

```
function puzzle(address _who, bytes32 _puzzle, bytes32 _emailHash)
only_owner {
    puzzles[_puzzle] = _emailHash;
    Puzzled(_who, _emailHash, _puzzle);
}
```

Figure 1: The puzzle implementation

After the user receives code, they can associate the email to an ethereum address by calling confirm ([ProofOfEmail.sol#L66-L77](#)):

```
function confirm(bytes32 _code) returns (bool) {
    var emailHash = puzzles[sha3(_code)];
    if (emailHash == 0)
        return;
    delete puzzles[sha3(_code)];
    if (entries[emailHash] != 0 || reverseHash[msg.sender] != 0)
        return;
    entries[emailHash] = msg.sender;
    reverseHash[msg.sender] = emailHash;
    Confirmed(msg.sender);
    return true;
}
```

Figure 2: The confirm implementation

Ethereum transactions are not accepted instantaneously. An attacker can observe the sent code before network consensus. Prior to the confirmation of the first transaction, they can submit the same code and associate the email with their own address. The attacker's

probability for success depends on the network state and the gas price for each transaction. An attacker can improve the odds of their transaction confirming first by offering a higher gas price.

As a result, the attacker can associate the victim's email address with the attacker's own ethereum address.

Another possible attack is an ethereum-address-to-email association denial-of-service. If the attacker's ethereum address is already associated, the code is deleted from the puzzles mapping ([ProofOfEmail.sol#L70-L72](#)) and future associations with this code will not be possible.

```
delete puzzles[sha3(_code)];  
if (entries[emailHash] != 0 || reverseHash[msg.sender] != 0)  
    return;
```

Figure 3: The confirm implementation (L70-L72)

Exploit Scenario

Bob wants to associate his email with his ethereum address. He requests the association and receives the validation code by email. He calls `confirm` with the validation code. Alice observes the transaction before the mining process completes. She calls `confirm` with the same validation code. Alice's transaction is mined before Bob's transaction. As a result, Alice's ethereum address is now associated with Bob's email.

Recommendation

Use the sender address from the request for the confirmation process.

Ensure that network latency from Ethereum transactions does not trigger unexpected behaviors.

References

- <https://ethgasstation.info/>

7. Incorrect interfaces may lead to unexpected behavior

Severity: Medium

Difficulty: Low

Type: Undefined Behavior

Finding ID: TOB-Parity-016

Target: ProofOfEmail.sol

Description

ProofOfEmail implements the wrong interface, impacting some of its functions.

ProofOfEmail inherits from the Certifier and ReverseRegistry contracts ([ProofOfEmail.sol#L42](#)):

```
contract ProofOfEmail is Owned, Certifier, ReverseRegistry {
```

Figure 1: The ProofOfEmail contract implementation

The Certifier and ReverseRegistry contracts are meant to be copies of Certifier.sol and Registry.sol ([ProofOfEmail.sol#L21-L40](#)):

```
// From Registry.sol
contract ReverseRegistry {
    event ReverseConfirmed(string indexed name, address indexed reverse);
    event ReverseRemoved(string indexed name, address indexed reverse);

    function hasReverse(bytes32 _name) constant returns (bool);
    function getReverse(bytes32 _name) constant returns (address);
    function canReverse(address _data) constant returns (bool) {}
    function reverse(address _data) constant returns (string) {}
}

// From Certifier.sol
contract Certifier {
    event Confirmed(address indexed reverse);
    event Revoked(address indexed reverse);

    function certified(address _who) constant returns (bool);
    function lookup(address _who, string _field) constant returns (string) {}
    function lookupHash(address _who, string _field) constant returns
(bytes32);
}
```

Figure 2: The ReverseRegistry and Certifier contract implementations

The code does not match the implementations present in Registry.sol ([Registry.sol#L37-L45](#)) and Certifier.sol ([Certifier.sol#L19-L26](#)):

```

contract ReversibleRegistry {
    event ReverseConfirmed(string indexed name, address indexed
reverse);
    event ReverseRemoved(string indexed name, address indexed
reverse);

    function hasReverse(bytes32 _name) constant returns (bool);
    function getReverse(bytes32 _name) constant returns (address);
    function canReverse(address _data) constant returns (bool);
    function reverse(address _data) constant returns (string);
}

```

Figure 3: The ReversibleRegistry contract implementation

```

contract Certifier {
    event Confirmed(address indexed who);
    event Revoked(address indexed who);
    function certified(address _who) constant public returns
(bool);
    function getData(address _who, string _field) constant public
returns (bytes32) {}
    function getAddress(address _who, string _field) constant
public returns (address) {}
    function getUint(address _who, string _field) constant public
returns (uint) {}
}

```

Figure 4: The Certifier contract implementation

We suspect that the ReversibleRegistry contract in Registry.sol is incorrectly named and should actually be named ReverseRegistry.

The ReverseRegistry in ProofOfEmail.sol implements canReverse and reverse as empty functions while they are interfaces in Registry's ReversibleRegistry contract. These functions are not overridden in the ProofOfEmail contract. As the result, the functions always return false.

The Certifier contract in ProofOfEmail.sol defines an empty lookup function. The ProofOfEmail contract implements the Certifier contract but does not override the lookup function. The lookup function is also absent from Certifier.sol's copy of the Certifier contract. Furthermore, the lookupHash function is present in ProofOfEmail.sol's copy of Certifier but not in Certifier.sol's copy.

The Certifier contract in Certifier.sol defines getData, getAddress, and getUint. These functions do not exist in ProofOfEmail.sol's copy of the Certifier contract. However, the ProofOfEmail contract in ProofOfEmail.sol defines a getAddress function.

These inconsistencies may lead to unexpected behavior for users and third-party programs.

Exploit Scenario

Bob creates a smart contract that uses ProofOfEmail. His smart contract performs some checks on data returned by lookup. Since lookup always returns 0, Bob's smart contract does not work.

Recommendation

Remove ReverseRegistry and Certifier from ProofOfEmail.sol. Import the contracts from Registry.sol and Certifier.sol instead. Clarify which functions should be present in each contract.

Use import instead of copying and pasting code.

Carefully review the interface of a contract.

8. Incorrect authorization prevents the calling of reporting functions

Severity: Medium

Difficulty: Low

Type: Access Controls

Finding ID: TOB-Parity-017

Target: RelaySet.sol

Description

An incorrect authorization schema prevents outerSet from calling the reportBenign and reportMalicious functions.

reportBenign and reportMalicious of outerSet call reportBenign and reportMalicious of innerSet, respectively ([RelaySet.sol#L71-L76](#)):

```
function reportBenign(address validator, uint256 blockNumber) public {
    innerSet.reportBenign(validator, blockNumber);
}

function reportMalicious(address validator, uint256 blockNumber, bytes
proof) public {
    innerSet.reportMalicious(validator, blockNumber, proof);
}
```

Figure 1: The reportBenign and reportMalicious implementations

These functions are public in outerSet.

reportBenign and reportMalicious in InnerOwnedSet are protected by the only_owner modifier ([InnerOwnedSet.sol#L116-L124](#)):

```
// Called when a validator should be removed.
function reportMalicious(address _validator, uint _blockNumber, bytes
_proof) public only_owner is_recent(_blockNumber) {
    Report(msg.sender, _validator, true);
}

// Report that a validator has misbehaved in a benign way.
function reportBenign(address _validator, uint _blockNumber) public
only_owner is_validator(_validator) is_recent(_blockNumber) {
    Report(msg.sender, _validator, false);
}
```

Figure 2: The reportBenign and reportMalicious implementations

`outerSet` is not meant to be the owner of `innerSet`. The call to `innerSet.reportBenign` and `innerSet.reportMalicious` will always fail. These failures cause `outerSet.reportBenign` and `outerSet.reportMalicious` to fail as well.

As a result, no one is able to call the reporting functions of `outerSet`.

Exploit Scenario

Bob wants to report Alice as a malicious user. Bob calls `outerSet.reportBenign`, but the call fails. As a result, Bob is not able to report Alice's malicious activity.

Recommendation

Fix the authorization schema of the `reportBenign` and `reportMalicious` functions of `InnerOwnedSet`. Consider changing the modifier `only_owner` to `only_outer` in `InnerOwnedSet.reportBenign` and `InnerOwnedSet.reportMalicious`.

Carefully review the authorization schema of the contracts. Create unit tests to ensure that each function can be called as expected.

9. “Unrequired” clients can remove a “required” client’s privilege

Severity: Medium

Difficulty: Low

Type: Data Validation

Finding ID: TOB-Parity-018

Target: Operations.sol

Description

Some clients of Operations have additional privileges and are considered “required.” A lack of validation in `setClientOwner` may allow an “unrequired” client to remove the privilege of a “required” client.

Only clients with `required` set to `True` can propose, confirm, or reject a transaction ([Operations.sol#L56-L58](#)):

```
struct Client {  
    address owner;  
    bool required;  
}
```

Figure 1: The Client Structure

Clients are represented by an address and a name. They can transfer their names to another address using `setClientOwner`, even if the address is already associated with a name ([Operations.sol#L169-L175](#)):

```
function setClientOwner(address _newOwner) only_client_owner {  
    var newClient = clientOwner[msg.sender];  
    clientOwner[msg.sender] = 0;  
    clientOwner[_newOwner] = newClient;  
    client[newClient].owner = _newOwner;  
    ClientOwnerChanged(newClient, msg.sender, _newOwner);  
}
```

Figure 2: The setClientOwner implementation

The client’s privilege is validated using its name ([Operations.sol#L290](#)):

```
modifier only_required_client_owner { var newClient =  
    clientOwner[msg.sender]; if (!client[newClient].required) throw; _; }
```

Figure 3: The only_required_client_owner implementation

An “unrequired” client can transfer its name to a “required” client. As a result, the “required” client will lose its privilege.

Exploit Scenario

Bob is a “required” client. Alice, an “unrequired” client, transfers her name to Bob by calling `setClientOwner` with Bob’s address. As a result, Bob loses his privilege.

Recommendation

Prevent a client from transferring their name to a client address already associated with a name.

Create a unit test for this functionality. A test on the name-transfer functionality may have found this vulnerability.

10. Missing contract existence check may cause unexpected behavior

Severity: Medium
Type: Data Validation
Target: RelaySet.sol

Difficulty: Medium
Finding ID: TOB-Parity-023

Description

OuterSet determines the current set of validators by calling its innerSet's `getValidators` function. The lack of a contract existence check allows calls to an invalid address to return successfully instead of throwing an error. This behavior causes the `getValidators` function to erroneously return an empty list of validators.

OuterSet.`getValidators` uses inline assembly code to call the innerSet.`getValidators` function ([RelaySet.sol#L59-L69](#)):

```
function getValidators() public constant returns (address[]) {
    address addr = innerSet;
    bytes4 sig = SIGNATURE;
    assembly {
        mstore(0, sig)
        let ret := call(0xffffffffface8, addr, 0, 0, 4, 0, 0)
        jumpi(0x02, iszero(ret))
        returndatacopy(0, 0, returdatasize)
        return(0, returdatasize)
    }
}
```

Figure 1: The `getValidators` implementation

The [Solidity documentation](#) warns:

The low-level `call`, `delegatecall`, and `callcode` will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

By default, `innerSet` is 0, which is a [valid address](#) that does not contain code. As a result, `getValidators` will return an empty list if `innerSet` is uninitialized or set incorrectly.

Exploit Scenario

Bob deploys two OuterSet contracts. Bob forgets to initialize the `innerSet` for one of the OuterSet contracts. Alice is a validator in the forgotten OuterSet. She deploys a smart contract that concatenates the list of validators from the two OuterSet contracts and uses it

for authorization. The `getValidator` function for the forgotten `OuterSet` contract returns an empty list. Alice wants to interact with her smart contract but is denied authorization.

Recommendation

Do not use inline assembly code. We did not find a valid reason for not using a high-level call, such as:

```
return innerSet.getValidators();
```

Fixing this issue may also fix issue [Github Issue #99](#).

Assembly code should only be used when absolutely needed. Using assembly code bypasses many of the additional checks provided by higher level Solidity primitives. If assembly code must be used, carefully review the [Solidity documentation](#) and the [Yellow Paper](#).

11. Race condition may lead to content compromise

Severity: Medium

Type: Timing

Target: GithubHint.sol

Difficulty: High

Finding ID: TOB-Parity-008

Description

GithubHint allows anyone to associate a hash to some external content (URL or GitHub commit). A race condition may allow an attacker to preempt a GithubHint transaction and associate the specified hash to malicious content.

In GithubHint.sol, hint and hintUrl associate a hash to some external content ([GithubHint.sol#L16-L22](#)):

```
function hint(bytes32 _content, string _accountSlashRepo, bytes20 _commit)
when_edit_allowed(_content) {
    entries[_content] = Entry(_accountSlashRepo, _commit, msg.sender);
}

function hintURL(bytes32 _content, string _url) when_edit_allowed(_content)
{
    entries[_content] = Entry(_url, 0, msg.sender);
}
```

Figure 1: The hint and hintURL implementations

hint and hintURL have the when_edit_allowed modifier. This modifier stops the transaction without throwing an error or returning a value if the hash has already been submitted ([GithubHint.sol#L14](#)):

```
modifier when_edit_allowed(bytes32 _content) { if (entries[_content].owner
!= 0 && entries[_content].owner != msg.sender) return; _; }
```

Figure 2: The when_edit_allowed modifier

Ethereum transactions are not instantaneously accepted. An attacker can observe the sent hash before network consensus. They can submit the same hash and associate it with malicious content using the hint functions prior to the confirmation of the first transaction. The attacker's probability for success depends on the network state and the gas price for each transaction. An attacker can increase the odds of their transaction confirming first by offering a high gas price.

Exploit Scenario

Bob uses GithubHint to associate an image with a hash in his decentralized application. Alice steals the ownership of his entry and associates it to malicious content. As a result, Alice is able to phish Bob's application.

Recommendation

Make hint and hintURL return a boolean denoting the call's status to the caller. Throw an error if the hash entry is already used.

Ensure that network latency from Ethereum transactions does not trigger unexpected behaviors.

References

<https://ethgasstation.info/>

12. Fork re-proposition may prevent owners from accepting or rejecting a fork

Severity: Low

Type: Data Validation

Target: Operations.sol

Difficulty: Medium

Finding ID: TOB-Parity-004

Description

A re-proposed fork cannot be confirmed or rejected by certain owners.

A fork can be proposed, accepted or rejected. Each fork is identified by its number. The fork structure contains the status mapping, which is used to store the decisions of the owners (undecided, accepted or rejected) ([Operations.sol#L70-L77](#)) :

```
struct Fork {  
    bytes32 name;  
    bytes32 spec;  
    bool hard;  
    bool ratified;  
    uint requiredCount;  
    mapping (bytes32 => Status) status;  
}
```

Figure 1: The Fork structure

An owner can only perform one action (accept or reject) and cannot undo their choice. Once a fork is rejected, it is deleted. The Solidity [documentation](#) states that:

delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

As a result, a fork deletion will not delete its status mapping.

If a fork is proposed with the same number, it will reuse the memory of the deleted fork. The new fork will also inherit the status of the deleted fork. As a result, all the owners that took part in the deleted fork decision will be unable to take part in the confirmation of the new fork.

Exploit Scenario

Bob is the only owner of `Operations` and proposes the fork 0. Bob then rejects the fork 0 and re-proposes a new fork, also with the number 0. Bob is now unable to accept or reject the new fork. In order to confirm the fork, Bob needs to add a new owner.

Recommendation

Prevent forks from being re-proposed.

Carefully review the [Solidity documentation](#), especially the memory model.

Create unit tests. A test on the rejection API could have found this vulnerability.

13. Owners cannot accept or reject re-proposed transactions

Severity: Low

Type: Data Validation

Target: Operations.sol

Difficulty: Medium

Finding ID: TOB-Parity-019

Description

A re-proposed transaction cannot be confirmed or rejected by certain owners.

A transaction can be proposed, accepted or rejected. Each transaction is identified by its number.

The transaction structure contains the status mapping, which is used to store the decisions of the owners (undecided, accepted or rejected) ([Operations.sol#L79-L86](#)):

```
struct Transaction {
    uint requiredCount;
    mapping (bytes32 => Status) status;
    address to;
    bytes data;
    uint value;
    uint gas;
}
```

Figure 1: The Transaction structure implementation

An owner can only perform one action (accept or reject) and cannot undo their choice. Once a transaction is accepted or rejected, it is deleted. The Solidity [documentation](#) states that:

delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

As a result, a transaction deletion will not delete its status mapping.

If a transaction is proposed with the same number, it will reuse the memory of the deleted transaction. The new transaction will also inherit the status of the deleted transaction. As a result, all the owners that took part in the acceptance or deletion of the original transaction will be unable to take part in the confirmation of the new transaction.

Note: this issue is similar to [TOB-Parity-004](#).

Exploit Scenario

Bob is the only owner of Operations and proposes a transaction. Alice accepts the transaction. Eve proposes a new transaction with the same number. Bob and Alice are now unable to accept or reject Eve's transaction.

Recommendation

Prevent transactions from being re-proposed.

Carefully review the [Solidity documentation](#), especially the memory model.

Create unit tests. A test on the rejection API could have found this vulnerability.

14. Lack of argument validation may lead to incorrect deletion of badge information

Severity: Low
Type: Data Validation
Target: BadgeReg.sol

Difficulty: High
Finding ID: TOB-Parity-025

Description

A lack of validation in `unregister` may lead to the incorrect deletion of information.

A Badge contains an address and a name ([BadgeReg.sol#L22-L24](#)):

```
struct Badge {  
    address addr;  
    bytes32 name;
```

Figure 1: The Badge structure

Badges are stored in the `badges` array. `mapFromAddress` and `mapFromName` mappings are used to determine the registration status of an address or a name, respectively ([BadgeReg.sol#L107-L109](#)):

```
mapping (address => uint) mapFromAddress;  
mapping (bytes32 => uint) mapFromName;  
Badge[] badges;
```

Figure 2: The `mapFromAddress`, `mapFromName` and `badges` variables

`unregister` deletes a badge by removing its address from `mapFromAddress` and removing its name from `mapFromName` ([BadgeReg.sol#L52-L57](#)):

```
function unregister(uint _id) only_owner public {  
    Unregistered(badges[_id].name, _id);  
    delete mapFromAddress[badges[_id].addr];  
    delete mapFromName[badges[_id].name];  
    delete badges[_id];  
}
```

Figure 3: The `unregister` implementation

If `unregister` is called on a badge that has already been deleted, the deletion occurs to address 0 and an empty string. This operation will occur even if address 0 or the empty

string already associates to a valid badge. As a result, the address or the name of the valid badge may be erroneously considered as unregistered.

Exploit Scenario

Alice registers a badge with an empty name. Bob, the owner of the contract, calls `unregister` on an already deleted badge. An empty name is now considered as unregistered. Eve registers a badge with an empty name. As a result, two badges with different addresses are registered with the same empty name.

Recommendation

Prevent `unregister` from being called on deleted badges.

If a badge is not supposed to be registered with address 0 or an empty name, prevent the registration in the `registerAs` function.

Consider checking function parameters for all unexpected values.

15. Deleting clients may lead to incorrect getter values

Severity: Low

Type: Data Validation

Target: Operations.sol

Difficulty: Low

Finding ID: TOB-Parity-005

Description

Getter functions return information on clients even after clients are deleted. This behavior may lead to unexpected behavior in third-party applications or contracts.

The Operations contract contains a client mapping consisting of client structures ([Operations.sol#313](#)).

```
mapping (bytes32 => Client) public client;
```

Figure 1: The client mapping

A client structure has three mappings ([Operations.sol#L56-L62](#)) :

```
struct Client {  
    address owner;  
    bool required;  
    mapping (bytes32 => Release) release;  
    mapping (uint8 => bytes32) current;  
    mapping (bytes32 => Build) build;           // checksum -> Build  
}
```

Figure 2: The Client structure

The client mappings can be accessed through the getters isLatest, track, latestInTrack, build, release and checksum ([Operations.sol#L226-L256](#)).

These mappings are not deleted when the client is deleted through removeClient ([Operations.sol#L199-L204](#)):

```
function removeClient(bytes32 _client) only_owner {  
    setClientRequired(_client, false);  
    resetClientOwner(_client, 0);  
    delete client[_client];  
    ClientRemoved(_client);  
}
```

Figure 3: The removeClient implementation

The solidity [documentation](#) states that:

delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

As a result, once a client is deleted, the getters continue to return the information of deleted clients. This behavior may cause unexpected behavior in third-party contracts.

This issue is similar to the issues [TOB-Parity-009](#), [TOB-Parity-010](#) and [TOB-Parity-011](#).

Exploit Scenario

Bob is a client in Operations. Bob is subsequently removed as a client. Despite being removed, Bob continues to appear as a client through the use of getters via a third-party.

Recommendation

Revert a call to a getter when the target is a deleted client.

Carefully review the [Solidity documentation](#), especially the memory model.

Create unit tests. A test on the [deleting API](#) could have found this vulnerability.

16. Deleting entries may lead to incorrect getter value (SimpleRegistry)

Severity: Low

Type: Data Validation

Target: SimpleRegistry.sol

Difficulty: Low

Finding ID: TOB-Parity-009

Description

Getter functions return information on entries even after entries are deleted. This behavior may lead to unexpected behavior in third-party applications or contracts.

The SimpleRegistry contract contains a entries mapping consisting of Entry structures ([SimpleRegistry.sol#L169](#)).

```
mapping (bytes32 => Entry) entries;
```

Figure 1: The entries mapping

An entry structure has a mapping field ([SimpleRegistry.sol#L47-L51](#)) :

```
struct Entry {  
    address owner;  
    address reverse;  
    mapping (string => bytes32) data;  
}
```

Figure 2: The Entry structure

The data mapping can be accessed through the getters getData, getAddress and getUint ([SimpleRegistry.sol#L57-L66](#)).

This mapping is not deleted when the entry is deleted through drop ([SimpleRegistry.sol#L94-L99](#)):

```
function drop(bytes32 _name) only_owner_of(_name) returns (bool  
success) {  
    delete reverses[entries[_name].reverse];  
    delete entries[_name];  
    Dropped(_name, msg.sender);  
    return true;  
}
```

Figure 3: The drop implementation

The Solidity [documentation](#) states that:

delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

As a result, once an entry is deleted, the getters continue to return the information of deleted entries. This behavior may cause unexpected behavior in third-party contracts.

This issue is similar to the issues [TOB-Parity-005](#), [TOB-Parity-010](#) and [TOB-Parity-011](#).

Exploit Scenario

Bob reserves an entry in SimpleRegistry. Bob's entry is removed. Despite being removed, Bob's entry continues to appear as a valid entry through the use of getter via a third-party.

Recommendation

Revert a call to a getter when the target is a deleted badge.

Carefully review the [Solidity documentation](#), specifically the memory model.

Create a unit test for this functionality. A test on the [deleting API](#) may have found this vulnerability.

17. Deleting dapps may lead to incorrect getter value (DappReg)

Severity: Low
Type: Data Validation
Target: DappReg.sol

Difficulty: Low
Finding ID: TOB-Parity-010

Description

Getter functions return information on dapps even after dapps are deleted. This behavior may lead to unexpected behavior in third-party applications or contracts.

The DappReg contract contains a dapps mapping consisting of Dapp structures ([DappReg.sol#L56](#)).

```
mapping (bytes32 => Dapp) dapps;
```

Figure 1: The dapps mapping

A Dapp structure has a mapping field ([DappReg.sol#L25-L29](#)):

```
struct Dapp {  
    bytes32 id;  
    address owner;  
    mapping (bytes32 => bytes32) meta;  
}
```

Figure 2: The Dapp structure

The meta mapping can be accessed through the getter meta(bytes32 _id, bytes32 _key) ([DappReg.sol#L93-L96](#)).

This mapping is not deleted when the dapp is deleted through unregister ([DappReg.sol#L87-L91](#)):

```
// remove apps  
function unregister(bytes32 _id) either_owner(_id) public {  
    delete dapps[_id];  
    Unregistered(_id);  
}
```

Figure 3: The unregister implementation

The solidity [documentation](#) states that:

delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

As a result, once a dapp is deleted, the getter continues to return the information of deleted dapps. This behavior may cause unexpected behavior in third-party contracts.

This issue is similar to the issues [TOB-Parity-005](#), [TOB-Parity-009](#) and [TOB-Parity-011](#).

Exploit Scenario

Bob registers a dapp in DappReg. Bob's dapp is removed. Despite being removed, Bob's dapp continues to appear as a valid dapp through the use of getter via a third-party.

Recommendation

Revert a call to a getter when the target is a deleted dapp.

Carefully review the [Solidity documentation](#), specifically the memory model.

Create unit test. A test on the [deleting API](#) could have found this vulnerability.

18. Deleting badges may lead to incorrect getter value (BadgeReg)

Severity: Low
Type: Data Validation
Target: BadgeReg.sol

Difficulty: Low
Finding ID: TOB-Parity-011

Description

Getter functions return information on badges even after badges are deleted. This behavior may lead to unexpected behavior in third-party applications or contracts.

The BadgeReg contract contains a badges array consisting of Badge structures ([BadgeReg.sol#L109](#)).

```
Badge[] badges;
```

Figure 1: The badges mapping

A Badge structure has a mapping field ([BadgeReg.sol#L22-L27](#)):

```
struct Badge {  
    address addr;  
    bytes32 name;  
    address owner;  
    mapping (bytes32 => bytes32) meta;  
}
```

Figure 2: The Badge structure

The meta mapping can be accessed through the getter meta(uint _id, bytes32 _key) ([BadgeReg.sol#L86-L88](#)).

This mapping is not deleted when the badge is deleted through unregister ([BadgeReg.sol#L52-L57](#)):

```
function unregister(uint _id) only_owner public {  
    Unregistered(badges[_id].name, _id);  
    delete mapFromAddress[badges[_id].addr];  
    delete mapFromName[badges[_id].name];  
    delete badges[_id];  
}
```

Figure 3: The unregister implementation

The Solidity [documentation](#) states that:

delete has no effect on whole mappings (as the keys of mappings may be arbitrary and are generally unknown). So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.

As a result, once a badge is deleted, the getter continues to return the information of deleted badges. This behavior may cause unexpected behavior in third-party contracts.

This issue is similar to the issues [TOB-Parity-005](#), [TOB-Parity-009](#) and [TOB-Parity-010](#).

Exploit Scenario

Bob registers a badge in BadgeReg. Bob's badge is removed. Despite being removed, Bob's badge continues to appear as a valid badge through the use of getter via a third-party.

Recommendation

Revert a call to a getter when the target is a deleted badge.

Carefully review the [Solidity documentation](#), specifically the memory model.

Create a unit test for this functionality. A test on the [deleting API](#) may have found this vulnerability.

19. Empty keyServerIp may lead to incorrect keyServersList

Severity: Informational

Difficulty: Low

Type: Data Validation

Finding ID: TOB-Parity-020

Target: key_servers_set.sol

Description

KeyServerSet keeps a list of server names associated with IP addresses. A server name can be associated only once. This assumption can be broken by associating a server name with an empty IP address.

addKeyServer associates a server name with an IP address ([key_servers_set.sol#L63-L74](#)):

```
// Add new key server to set.
function addKeyServer(bytes keyServerPublic, string keyServerIp)
public only_owner valid_public(keyServerPublic)
new_key_server(computeAddress(keyServerPublic)) {
    // compute address from public
    address keyServer = computeAddress(keyServerPublic);
    // fire event
    KeyServerAdded(keyServer);
    // append to the list and to the map
    keyServers[keyServer].index = keyServersList.length;
    keyServers[keyServer].publicKey = keyServerPublic;
    keyServers[keyServer].ip = keyServerIp;
    keyServersList.push(keyServer);
}
```

Figure 1: The addKeyServer implementation

keyServersList stores all the associations.

addKeyServer has the new_key_server modifier to ensure that the server name is not currently associated ([key_servers_set.sol#L43-L44](#)):

```
// Only run if server is not currently in the set.
modifier new_key_server(address keyServer) {
    require(sha3(keyServers[keyServer].ip) == sha3(""));
    _;
}
```

Figure 2: The new_key_server implementation

If a server name is associated with an empty-string IP address, `new_key_server` will return `true`. As a result, `keyServersList` can contain multiple associations for the same server name.

Recommendation

Ensure that `keyServerIp` is not an empty string.

Consider checking function parameters for all unexpected values.

20. Contracts specify outdated compiler version

Severity: Informational

Type: Patching

Target: All smart contracts

Difficulty: Undetermined

Finding ID: TOB-Parity-026

Description

Some Parity contracts specify an outdated version of the Solidity compiler.

The Solidity compiler is under active development. Each new version contains new checks and warnings for suspect code.

- SMT Checker: Take if-else branch conditions into account in the SMT encoding of the program variables.
- Syntax Checker: Deprecate the `var` keyword (and mark it an error as experimental 0.5.0 feature).
- Type Checker: Allow `this.f.selector` to be a pure expression.
- Type Checker: Issue warning for using `public` visibility for interface functions.
- Type Checker: Limit the number of warnings raised for creating abstract contracts.

Figure 1: Solidity releases new checks and warnings for suspect code in each new version

Running the latest available compiler (0.4.20 as of this writing) on the Parity contracts codebase emits warnings that should be fixed.

There are also inconsistencies in the Solidity compiler version requirements between contracts and their dependencies. For example, [SimpleCertifier.sol](#) requires Solidity 0.4.7 while [Owned.sol](#), a contract which [SimpleCertifier.sol](#) imports, requires Solidity 0.4.17.

Recommendation

Ensure that the latest version of Solidity compiles all code without warnings. Compiler warnings are often indicators of bugs that may only manifest at runtime or under specific conditions. Newer versions of Solidity emit warnings for a broader set of error-prone programming practices.

Standardize the version of Solidity required by contracts and their dependencies.

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
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

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Smart Contracts General Recommendations

- Make the visibility of functions explicit. This would prevent mistakes regarding a function's scope.
- Use `import` instead of copying and pasting code. It is difficult to keep copied code up-to-date.
- When possible, use [interface](#) instead of [abstract contract](#). Interface makes the inheritance clearer.
- Use `require` in modifiers instead of if-then-else structures. `require` makes the code more readable.
- Ensure that comments in the source code are in sync with the behavior of the source code.

Operations.sol

- Do not shadow built-in Solidity variables such as `now` and `tx`. The `now` variable is shadowed in `ClientOwnerChanged`, `ClientRequiredChanged` and `OwnerChanged`. The `tx` variable is shadowed in `checkProxy`.
- Be explicit in the documentation about the behavior of `Operations` when there is only one required client. In this case, all the transactions are directly executed when proposed and there is no need to call the confirmation function. Incorrect usage may result from this undocumented behavior.

SMSVerification.sol and SimpleCertifier.sol

- Remove duplicate contracts. The version of the `SimpleCertifier` contract in `SMSVerification.sol` is slightly different than the version of the `SimpleCertifier` contract in `SimpleCertifier.sol`. It includes visibility modifiers and has been updated to use `require()`. Consider consolidating these into one implementation to prevent future desynchronization and the potential for future unexpected behaviors.
- Clarify the expected behavior of `SimpleCertifier` in case of failure. `SMSVerification.SimpleCertifier` and `SimpleCertifier.SimpleCertifier` are meant to be equivalent, but differ in case of failure: `SMSVerification.SimpleCertifier` throws an error while `SimpleCertifier.SimpleCertifier` returns without error.

SimpleRegistry.sol

- Do not shadow built-in Solidity keywords such as `reserved`.

ProofOfEmail.sol

- Use a non-zero value for `fee` to avoid triggering events for free during contract deployment.

Configuration Recommendations

- Consider setting more restrictive permissions by removing `group` and `world` access from `vault.json`.

C. Slither

Trail of Bits has included our Solidity static analyzer, Slither, with this report. Slither works on the Abstract Syntax Tree (AST) generated by the Solidity compiler and detects some of the most common smart contract security issues including:

- The lack of a constructor
- The presence of unprotected functions
- Uninitialized variables
- Unused variables
- Functions declared as constant that change the state
- Deletion of a structure containing a mapping

Slither is an unsound static analyzer and may report false positives. The lack of proper support for inheritance and some object types (such as arrays) may lead to false positives.

Usage

Launch the analysis on the Solidity file:

```
$ python /path/to/slither.py file.sol
```

Example

In the example below, Slither found that the structure `Badge` in the contract `BadgeReg` is deleted in `unregister` but contains a mapping. This corresponds to [TOB-Parity-011](#).

```
$ python slither.py BadgeReg.sol
INFO:Slither:Deletion on struct with mapping in
parity/BadgeReg.sol.ast.json, Contract: BadgeReg, Func/Struct:
[(u'unregister', u'Badge')]
```

Discovered Issues in Parity Code

Slither found issues in the Parity codebase that – due to time constraints – we did not fully investigate to determine their validity. These issues included:

- Incorrect constructor name in [SignedReceiver](#)
- Deletion of a structure containing a mapping in [TokenReg](#)
- [tokenRecorder](#) is [used](#) but never assigned in [DutchBuyin](#)
- [Record](#) is [used](#) but never assigned in [FairReceiver](#)

D. Fix Log

Trail of Bits performed a retest of the Parity system during the weeks of June 11 and July 2, 2018. Parity provides fixes and supporting documentation for the findings outlined in their most recent security assessment report. Each of the findings was re-examined and verified by Trail of Bits.

Prior to these reviews, Trail of Bits recommends adopting development practices that clearly identify when and how bugs have been fixed. In particular, developers should:

- **Fix each bug in a separate commit.** Do not merge unrelated modifications into one commit. It makes reviewing the changes more difficult.
- **Use git submodules.** Do not duplicate code. Duplicated code is error-prone and makes updating the code more difficult. For example Owned.sol is shared [secretstore-key-server-set](#), [kovan-validator-set](#), [dapp-registry](#) and [name-registry](#).
- **Keep track of fixed issues.** If you use Github to track fixes, mark the issue as resolved once it is fixed by using the built-in “[Fixed](#)” syntax.

Parity reorganized the cryptographic module into ethcore-crypto and fully addressed four issues, partially addressed three issues, and did not address one issue in the Rust code. In particular, the majority, but not all, of the cryptographic code was ported from rust-crypto to [ring](#). Trail of Bits recommend future review of ethcore-crypto due to these recent modifications.

Parity reorganized the smart contracts and split them into separate repositories. All the identified smart contract issues were fully addressed or the affected contract was deprecated. The new method of organizing the contracts more clearly identifies its scope:

- KeyServerSet → <https://github.com/parity-contracts/secretstore-key-server-set>
- Operations (auto-update) → <https://github.com/parity-contracts/auto-updater>
- Badge contracts (sms/email verification) → deprecated
- Dapp <https://github.com/parity-contracts/dapp-registry/>
- Urlhint → <https://github.com/parity-contracts/github-hint>
- RelaySet → <https://github.com/parity-contracts/kovan-validator-set>
- InnerOwnedSet → <https://github.com/parity-contracts/kovan-validator-set>
- SimpleRegistrar → <https://github.com/parity-contracts/name-registry>
- SimpleCertifier → <https://github.com/parity-contracts/name-registry>

Rust Fix Log

#	Title	Severity	Status
1	Key files may be deleted without authorization during wallet import	High	Fixed
2	Rust-crypto is not recommended for security-critical usage	High	Partial fix (*)
3	HMAC comparison in do_decrypt is vulnerable to timing attacks	Medium	Fixed
4	"single message" crypto operations lack authentication due to using AES-CTR	Medium	Fixed
5	Deserialized address field in SafeAccount is not properly sanitized	Medium	Not fixed (**)
6	Content-Security-Policy is overly permissive	Low	Partial fix (***)
7	Confidential information resides in memory for too long	Low	Fixed
8	Parity executables on Windows lack code signatures	Low	Partial fix (****)

(*) See [TOB-Parity-028](#) Fix

(**) See [TOB-Parity-021](#) Fix

(***) See [TOB-Parity-001](#) Fix

(****) See [TOB-Parity-028](#) Fix

Solidity Fix Log

#	Title	Severity	Status
1	Re-entrancy may lead to stolen ethers	High	Fixed
2	Missing loop iteration leads to non-removable validator	Medium	Fixed
3	Incorrect interface implementation leads to unexpected behavior	Medium	Fixed
4	Incorrect conditional prevents fork rejection	Medium	Fixed
5	Uninitialized value leads to an unmodifiable owners list	Medium	Fixed
6	Race condition may preempt an Ethereum address to email association	Medium	Deprecated
7	Incorrect interfaces may lead to unexpected behavior	Medium	Deprecated
8	Incorrect authorization prevents the calling of reporting functions	Medium	Fixed
9	"Unrequired" clients can remove a "required" client's privilege	Medium	Fixed
10	Missing contract existence check may cause unexpected behavior	Medium	Fixed
11	Race condition may lead to content compromise	Medium	Fixed
12	Fork re-proposition may prevent owners from accepting or rejecting a fork	Low	Fixed
13	Owners cannot accept or reject re-proposed transactions	Low	Fixed
14	Lack of argument validation may lead to incorrect deletion of badge information	Low	Deprecated
15	Deleting clients may lead to incorrect getter values	Low	Fixed

16	Deleting entries may lead to incorrect getter value (SimpleRegistry)	Low	Fixed
17	Deleting dapps may lead to incorrect getter value (DappReg)	Low	Fixed
18	Deleting badges may lead to incorrect getter value (BadgeReg)	Low	Deprecated
19	Empty keyServerIp may lead to incorrect keyServersList	Informational	Fixed
20	Contracts specify outdated compiler version	Informational	Fixed

Detailed Issue Discussions

In this section, we note reasons why certain issues are labeled “Unfixed” or “Partial Fix.” Responses from Parity about each outstanding issue are included as quotes.

TOB-Parity-001 Fix

`unsafe-eval` is still present in some Dapps.

Won't fix. Some dapps require ``eval`` to provide their functionality, e.g. web3 console. We've made some changes so that dapps must explicitly require ``unsafe-eval`` in their manifest.

TOB-Parity-021 Fix

Fixing this issue necessitates a change in the user interface that was not made.

Won't fix. This is mostly outside of our security model since it involves access to the local hard drive, furthermore the fix would require UI changes and we are reducing our investment on the UI (in the latest releases it is no longer part of the binary).

TOB-Parity-027 Fix

The uninstaller is not included in the list of executables to be signed.

The only Windows binary currently not signed is the uninstaller since it will be removed in an upcoming release (expected lifetime of the uninstaller is another 8 weeks).

TOB-Parity-028 Fix

Refactored cryptographic components still use [rust-crypto](#).

We have replaced some usages with implementations from ``ring`` but it doesn't provide replacements for all usages from ``rust-crypto``. The alternative would be to depend on a C/C++ crypto library but that goes against our goals of eventually compiling parity to WebAssembly, and it is very likely that Rust bindings would be unaudited. For now we will continue the existing work of migrating to ``ring`` and we will consider other Rust crypto implementations as they appear. Eventually having ``ring`` (or other crypto library we may rely on) audited is also a possibility.

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, blockchain, and software exploits. According to their needs, we may review 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.

Find out more about Trail of Bits on our [website](#) and [blog](#).