Ethereum Clef Review

Ethereum Foundation

September 14, 2018 – Version 1.0

Prepared for
Martin Swende

Prepared by
David Wong
Eric Schorn

Executive Summary

In the start of September 2018, the Ethereum Foundation contracted NCC Group to perform a review of the Clef command-line interface. The Clef CLI is a self-contained account management tool that can also be used as a "signing server" to auto-approve transactions and other available methods of an API exposed through various interfaces. One consultant was tasked to look for usability and security issues with the tool and its API. One extra consultant was added to the project at no extra cost to the Ethereum Foundation. A discord channel was used to communicate with the development team. Some medium-risk issues were discovered while a few observations and recommendations were written up. No major issues were found.

Scope

Commit 70cfedc9d7bd64f1f112eb2099a5c36266863f40 of Clef was audited, the scope included the following items:

Clef. The Ethereum Foundation has developed a self-contained tool for account management. It can be used to create accounts, list them, sign transactions, sign arbitrary data and recover public keys from signatures. It can be used as a simple command-line interface (CLI) or with a graphical user interface (GUI). Requests are made through an API exposed via an IPC or a JSON-RPC interface. Clef can also be used with different types of "backends", from simple file system wallets to hardware wallets like the Trezor[1] and the Nano Ledger.[2]

Rules. Clef supports automation of requests handling, allowing users to develop Javascript functions that will approve or reject requests to the API based on time, the Clef state and the request being made. This allows Clef to run without user-interaction (although it will be required as a fallback if the functions written fail to correctly approve or reject requests).

The following items were not included in scope:

Crypto. Clef relies on a couple of cryptographic primitives which are not part of the standard go library: keccak256 (a variant of SHA-3) and secp256k1 (for the ECDSA signature algorithm and recovering public keys out of such signatures).

Accounts. Clef relies heavily on the `accounts` package of go-ethereum for parsing a transaction's call data via `accounts/abi`, for managing accounts via `accounts/keystore` and for compatibility with hardware wallets

via `accounts/usbwallet`. While a full review of these packages was not in scope, the consultants have partially reviewed them when relevant to the Clef application.

BigNumber.js. Javascript rules written for Clef can make use of this extra dependency for handling big numbers. This library is a single file of less than 3000 lines of code and was not audited as part of this engagement.

Otto. Otto is a Javascript interpreter written in Go used by Clef to execute the rules written by users. It is quite an important package with around 40,000 lines of code. Note that this package is also used in other parts of go-ethereum.

GUI-based Clef. Clef can be used in conjunction with a graphical interface. The consultants did not spend time looking at these applications.

Key Findings

While no major findings were found, a few medium-risk findings were discovered:

- **Lack of Complexity Check for Passphrases**. As it is, Clef does not enforce any minimum-length on passphrases used to protect private keys. Users are incentivized not to use a passphrase (length 0) or to use a weak passphrase (length < 8).
- **Denial of Services of Clef's API**. Malformed requests can crash the application, which could be of temporary damage to long-lived automated configurations of Clef.
- **Encryption of Clef Backup is Insufficient**. Clef uses a master secret to encrypt several important piece of information including passphrases that can unlock Ethereum wallets. This master secret is then stored in clear on the device running Clef, accessible to users with enough permissions, to physical breaches on devices with no disk encryption or to accidental copies of the file to other locations.

Strategic Recommendations

NCC Group recommends that the Ethereum Foundation takes the following points into consideration in order to increase the security stance of Clef:

- **Documentation**. Thoroughly document the threat model of Clef, and list what users need to protect against in high-stake situations. Ensure that all examples of rules are up-to-date and secure by default

---

[1] https://trezor.io/
[2] https://www.ledger.com/

so that they can be copy/pasted by users.

- **External Dependencies**. Clef makes use of large dependencies that could impact the well-functioning of the program. Otto[3] is used as a Javascript interpreter to run rules written by users; bignumber.js[4] is used as a Javascript library to handle large numbers. The Ethereum Foundation should ensure that these dependencies are up-to-date and have been audited.

- **Go-Ethereum Dependencies**. The `accounts` package is a large piece of go-ethereum that Clef leverages. It handles on-disk encryption and storage of wallets, communication with hardware wallets, and contracts' ABI logic. The `crypto` package is used by Clef to handle public-key cryptography logic and hashing. The Ethereum Foundation should consider auditing these parts as they are central pieces of the tool.

---

[3] https://github.com/robertkrimen/otto
[4] https://github.com/MikeMcl/bignumber.js/

Dashboard

nccgroup

Target Metadata

Name	Clef
Type	Command-Line Interface
Platforms	Go
Environment	Local Instance

Engagement Data

Type	Code Review
Method	Code-assisted
Dates	2018-09-03 to 2018-09-14
Consultants	1
Level of effort	10 person-days

Targets

Clef	github.com/holiman/go-ethereum/blob/clefchanges_2/cmd/clef

Finding Breakdown

Critical Risk issues	0
High Risk issues	0
Medium Risk issues	4
Low Risk issues	8
Informational issues	1
Total issues	**13**
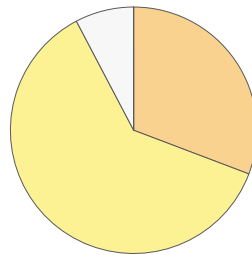
Category Breakdown

Authentication	2	
Configuration	2	
Cryptography	3	
Data Validation	4	
Denial of Service	1	
Patching	1	

Component Breakdown

keystore	1	

Key

Critical		High		Medium		Low		Informational	

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see .

Title	ID	Risk
Encryption of Clef Backup is Insufficient	002	Medium
Lack of Password Strength Check	005	Medium
Validation Of Transaction Data Field Fails Open	007	Medium
Denial of Service Through Incorrect Method Selector	010	Medium
Incorrect File Permissions For `secrets.dat`	001	Low
Encrypted Values From Key-Value Encrypted Storage Are Swappable	003	Low
Lack of Guidance on Exposed Clef API	004	Low
ECRecover Does Not Authenticate The Recovered Public Key	009	Low
Outdated Dependencies	011	Low
Rules Dangerously Rely On Time And State	012	Low
Denial of Service Through Malformed Import Key	013	Low
Encrypted KeyStore Integrity Check Is Incomplete	014	Low
UI Mixes Extraneous and Approval-Specific Data	006	Informational

Finding Details

Finding	**Encryption of Clef Backup is Insufficient**
Risk	**Medium** Impact: High, Exploitability: Low
Identifier	NCC-EF-Clef-002
Category	Authentication
Location	• cmd/clef/main.go:initializeSecrets()
Impact	An attacker able to compromise a user's device has full access to Clef's encrypted back up.
Description	Clef stores a number of files on disk, such as `credentials.json` containing account credentials and `config.json` containing the JavaScript rule-file hashes, in order to be able to recover from a restart. These files are encrypted by keys derived from a single seed which is stored in cleartext on disk in the `secrets.dat` file. An attacker able to access `secrets.dat` and `credentials.json` would have full and unrestricted access to accounts and their value. Because of this, a device not protected with disk encryption that gets physically breached would leak all information stored under Clef's encrypted backup. A running device that gets remotely breached would also compromise a dormant Clef application, irrespective of disk encryption. Furthermore, an accidental non-encrypted copy or backup of these files to a different location could compromise a user's accounts.
Recommendation	Enforce usage of a passphrase to start the Clef command-line interface. Leverage the same passphrase mechanisms as in `go-ethereum/accounts/keystore` to protect the `secrets.dat` file.

Finding	**Lack of Password Strength Check**
Risk	**Medium** Impact: Medium, Exploitability: Low
Identifier	NCC-EF-Clef-005
Category	Authentication
Location	• signer/core/api.go:New()
Impact	An attacker may guess insecure user passwords or brute-force weak user passwords in the event of a breach. This would allow the attacker to retrieve private keys of the user's Ethereum accounts.
Description	The Clef CLI can be used to create and manage Ethereum externally owned accounts. The account creation process can be started by sending the following RPC request to Clef (if run with the `--rpc` option):

```
curl -H "Content-Type: application/json" -X POST --data '{"jsonrpc":"2.0","metho
→  d":"account_new","params":["test"],"id":67}' localhost:8550
```

	The user running the Clef process is then prompted to enter a password which is used to protect the account's private key. At this point the user can enter an arbitrary-length password (empty passwords are also accepted). This would facilitate recovery of the accounts's private keys to attackers physically or remotely breaching the device where Clef is installed.
Recommendation	Enforce a minimum password length. The NIST organization has published documents[5] about the topic, recommending to set a minimum-length of 8 characters for such passwords. Additionally, check for known bad passwords. Various lists of known bad passwords like the NBP[6] exist.
Client Response	Clef now enforce passwords of 10 characters at a minimum: github.com/holiman/go-ethereum/commit/193f7049719a2da9018027853d0c2237cdad602b

[5]https://pages.nist.gov/800-63-3/
[6]NIST Bad Passwords:https://cry.github.io/nbp/

Finding	**Validation Of Transaction Data Field Fails Open**
Risk	**Medium** Impact: High, Exploitability: Low
Identifier	NCC-EF-Clef-007
Category	Data Validation
Location	• signer/core/{abihelper,validation}.go
Impact	A maliciously crafted data field could allow an attacker to deceive the signer's intent.
Description	When receiving a request to sign a transaction, Clef first attempts to perform a few validation checks before passing the request to the user. If a special argument is passed to the request (a method's signature) the program also attempts to match it against the data field. In that case, the data field must be composed of a 4-byte identifier for the function called (which is the hash of the method's signature truncated to 4 bytes) and a multiple of 32 bytes:

```go
func parseCallData(calldata []byte, abidata string) (*decodedCallData, error) {
        if len(calldata) < 4 {
                return nil, fmt.Errorf("Invalid ABI-data, incomplete method sign
➥   ature of (%d bytes)", len(calldata))
        }
        sigdata, argdata := calldata[:4], calldata[4:]
        if len(argdata)%32 != 0 {
                return nil, fmt.Errorf("Not ABI-encoded data; length should be a
➥   multiple of 32 (was %d)", len(argdata))
        }
```

This check is not enforced if no method signature is passed as argument in the request to Clef. This is because method signatures are not an Ethereum Virtual Machine feature but a Solidity-specific feature. If the check fails, Clef still end up passing the request to the user with a warning:

```go
info, err = testSelector(*methodSelector, data)
if err != nil {
        msgs.warn(fmt.Sprintf("Tx contains data, but provided ABI signature coul
➥   d not be matched: %v", err))
}
```

Since users of the Clef CLI are not expected to always pass a method signature, or to understand the warning associated to a failed ABI signature check, the behavior of Clef might incentivize users to click through them (this is called alert fatigue). Because of this, malicious DAPPs could attempt short address attacks[7] or other yet unknown attacks where transactions' calldata is malformed.

Recommendation	In the cases where a method signature is passed, the data field format should always be enforced to be of length $4 + k \times 32$ with $k \geq 0$. If this is not the case, Clef should not pass the request to the end user. In the cases where a method signature is not passed and the data field is not empty, its format should still be checked against the previously discussed encoding. If it does not validate, Clef should reject the transaction (unless configured to lighten its validations or with a whitelist of relaxed contract addresses). Alternatively, if non-standard transactions need default support, the user should be warned that the transaction data field is not standard.

[7] https://www.dasp.co/#item-9

Client Response	Validations that return warnings are now rejecting transactions by default, a new "advanced" mode was added to bypass this behavior: github.com/holiman/go-ethereum/commit/193f7 049719a2da9018027853d0c2237cdad602b

Finding	**Denial of Service Through Incorrect Method Selector**
Risk	**Medium** Impact: Low, Exploitability: Low
Identifier	NCC-EF-Clef-010
Category	Data Validation
Location	• signer/core/abihelper.go:parseCallData() • accounts/abi/abi.go:JSON()
Impact	An attacker with access to Clef's API can crash the application.
Description	In some use-cases Clef is used to run continuously, accepting requests and accepting them based on rules written by the user. In such cases, a crash could prevent legitimate transactions to be processed until the application is restarted.

The `account_signTransaction` API handles transaction signing requests. In order to provide useful information to the user, the endpoint making the request can provide the method signature of the function being called (in cases where the transaction would result in a contract execution). If this method signature is malformed, Clef crashes. Currently a single regex is used to validate this user input:

```
// MethodSelectorToAbi converts a method selector into an ABI struct. The return
➝  ed data is a valid json string
// which can be consumed by the standard abi package.
func MethodSelectorToAbi(selector string) ([]byte, error) {
        re := regexp.MustCompile(`^([^\)]+)\(([a-z0-9,\[\]]*)\)`)
        groups := re.FindStringSubmatch(selector)
```

While the regex is expected to validate typical function signatures:

```
functionName(uint256, string, address)
```

It is too liberal, using a blacklist instead of a whitelist. This overly-accepting policy permits the following kind of user inputs:

- functionName can be anything but \ and)
- arguments can be alphanumeric strings and contain [and] but do not have to enforce syntactically correct brackets
- argument list can end and start with ,
- the end of the function signature can contain anything

This mean that the following function signatures are valid according to the current checks:

```
call(a,a],bbbb932[,)
#@#((@$!(uint256) anything
```

Reproduction Steps	Run the following in the terminal with a Clef process exposing an RPC interface on `local‐host:8550` and observe that the Clef application crashes.

```
curl -i -H "Content-Type: application/json" -X POST --data '{"jsonrpc":"2.0","me
 ➜  thod":"account_signTransaction","params":[{"from":"0x82A2A876D39022B3019932D
 ➜  30Cd9c97ad5616813","gas":"0x333","gasPrice":"0x123","nonce":"0x0","to":"0x07
 ➜  a565b7ed7d7a678680a4c162885bedbb695fe0", "value":"0x10", "data":"0x4401a6e40
 ➜  0000000000000000000000000000000000000000000000000000000000000012"}, "func(uin
 ➜  t256,uint256,[]uint256)"],"id":67}' http://localhost:8550/
```

The following method signatures all make the application crash:

```
func(uint256,uint256,[]uint256)
func(uint256,uint256,uint256,)
func(,uint256,uint256,uint256)
```

Recommendation | In order to address this issue:

1. Investigate the JSON decoder of the abi package to find the root cause of the error.
2. Further validate the received method signature before attempting to operate on it.

Client Response | A Pull Request was introduced to fix the bug: github.com/ethereum/go-ethereum/pull/17653

Finding	**Incorrect File Permissions For `secrets.dat`**
Risk	**Low** Impact: High, Exploitability: Low
Identifier	NCC-EF-Clef-001
Category	Configuration
Location	• Permissions set in `initializeSecrets()` on line 228 of `cmd/clef/main.go` • Permissions checked in `checkFile()` on line 550 of `cmd/clef/main.go`
Impact	The master seed may be deleted or overwritten resulting in the loss of access to account credentials and JavaScript rule-file hashes.
Description	The `secrets.dat` file contains the master seed, which is required to be able to store and retrieve account credentials and JavaScript rule-file hashes. In practice, this central file is written once, contains the critical root secret stored in the clear, and must be maximally protected. Master seed generation and storage is the primary purpose of the `initializeSecrets()` function in `cmd/clef/main.go`. The seed is written to disk on line 228 with file permissions set to `700`. This corresponds to full permissions for the owner – read, write and execute. As a result, the owner may easily delete or overwrite this file resulting in loss of access to the storage mentioned above. In principle, the owner may also attempt to execute this file. The primary purpose of the `checkFile()` function in `cmd/clef/main.go` is to check the file permissions of the `secrets.dat` file. On line 550, the file permissions are read, logically 'ANDed' with `077` and compared to `0` - with any result other than `0` being an error. This is consistent with verifying the storage permissions set in `initializeSecrets()` as described above. For `secrets.dat`, the write permission should not be set by default and the execution permission is also inappropriate. The handling of the `secrets.dat` file is analogous to handling SSH keys.[8] Separately, note that the file permissions for the account credentials stored in `credentials.json` and the JavaScript rule-file hashes stored in `config.json` are currently set to `600` by the `writeEncryptedStorage` function in `signer/storageaes_gcm_storage.go`. This is considered appropriate due to the read/write nature of the key-value storage and the fact that the contents are always encrypted by the root secret.
Recommendation	The file permissions for `secrets.dat` should be set to `400` (instead of `700`) in `initializeSecrets()`. The file permissions for `secrets.dat` should be 'ANDed' with `377` (instead of `077`) in `checkFile()` to maintain consistency.

[8] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/TroubleshootingInstancesConnecting.html#troubleshoot-unprotected-key

Finding	**Encrypted Values From Key-Value Encrypted Storage Are Swappable**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-EF-Clef-003
Category	Cryptography
Location	• signer/storage/aes_gcm_storage.go
Impact	An attacker with permissions to the encrypted backup files of Clef could swap around the encrypted passwords for the user's keystores. This could allow weak attacks (like confirming if different keys are protected by the same passphrase), or yet unknown complex attacks depending on the rules in use by Clef.
Description	The Clef command-line interface stores the following data on disk, in an encrypted form, in order to facilitate recovery after restarts of the application:

• passwords for keystores (used by rule engine)
• storage for javascript rules
• hash of rule-file

The storage and encryption is done via a key-value store where only the values are encrypted via AES-GCM:

```go
// Put stores a value by key. 0-length keys results in no-op
func (s *AESEncryptedStorage) Put(key, value string) {
    // ...
    ciphertext, iv, err := encrypt(s.key, []byte(value))
    // ...
    encrypted := storedCredential{Iv: iv, CipherText: ciphertext}
    data[key] = encrypted
    // ...
}
```

The key-values are then encoded in the JSON format and saved on disk as can be seen in the following example:

```json
{
  "key1": {
    "iv": "IQZYrnH0YjbcLmBD",
    "c": "oP2S7Li+YYPt2vQcfDgUlc/QaIk="
  },
  "key2": {
    "iv": "OVilp+zm+OvgH7Vm",
    "c": "DP7kmTyJR89nTMb1mfRPokIYRpg="
  }
}
```

An attacker with the correct permissions to these files can tamper them to swap around the values of `key1` and `key2` such that when `key1` is retrieved from storage, the value associated to `key2` is obtained.

Recommendation	Include the key part of the key-value in the `additionalData` field of the `Seal()` and `Open()` functions. See the cipher package[9] for more information.
Client Response	the key part of the key-value was added as additional data to AES-GCM: github.com/holiman

---

[9]https://golang.org/pkg/crypto/cipher/

/go-ethereum/commit/913f77ca8c5c08749b9d668adeb1ab02bbc30663

Finding	**Lack of Guidance on Exposed Clef API**
Risk	Low Impact: Low, Exploitability: Undetermined
Identifier	NCC-EF-Clef-004
Category	Denial of Service
Location	• cmd/clef/main.go
Impact	An attacker with access to the Clef API can quickly spam the interface and render it useless, forcing the user to restart the application in order to process legitimate requests.
Description	An attacker who has access to the public API of Clef (through an RPC interface exposed to the internet for example) can quickly spam the process with requests that will need to be manually handled, in order, by the end user.

If such an attack is performed, the end user will be incapable of going on with normal operations without restarting the process.

In addition, requests done over Ethereum's RPC protocol are not encrypted. While most of the API requests and responses might eventually be published on the Ethereum network, the "account_sign" method (aimed at signing arbitrary data for different purposes) might require secrecy.

Reproduction Steps

Run the following bash script with a Clef process exposing an RPC interface on `localhost:8550` and observe that the Clef user now has to accept requests one by one.

```
for i in {1..100}
do curl --no-buffer -H "Content-Type: application/json" -X POST --data '{"jsonrp
↪   c":"2.0","method":"account_new","params":["test"],"id":67}' localhost:8550 &
done
kill $(jobs -p)
```

Recommendation

Add ways to encrypt the connection (via TLS) and to authenticate clients to the Clef API.

Alternatively, delegate these tasks to a lower-layer protocol or a fronting proxy, but add documentation to warn users against the dangers of exposing Clef's API outside of their own machine.

Finding	**ECRecover Does Not Authenticate The Recovered Public Key**
Risk	**Low** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-EF-Clef-009
Category	Cryptography
Location	• signer/core/api.go
Impact	Depending on the usage of this request, signatures could be tampered with in order to recover the wrong public key.
Description	The Clef API exposes an `EcRecover` method that allows to recover an Ethereum public-key from a signed message. The method implements algorithm 4.1.6 (Public Key Recovery Operation) from the Standards for Efficient Cryptography Group document on Elliptic Curve Cryptography.[10]
	As noted by the algorithm's specification, several public keys can be recovered from a signature. This is due to the ECDSA signature algorithm removing some information from the `r` value of a signature: only the x-coordinate is retained (2 solutions can be recovered for the y-coordinate) and it is further reduced modulo the order of the elliptic curve. In the case of Ethereum, secp256k1 is used which has a curve order slightly lower than the field modulus, so indeed information is lost. The curve uses a cofactor of 1, so the number of possible solutions to the algorithm are $2 \times (1 + 1) = 4$.
	In order for the recovery algorithm to recover the correct solution, a `v` byte is added at the end of every Ethereum signature. Its least significant bit contains the sign of the y-coordinate of the `r` value and the rest of the bits contain information to re-compute the x-coordinate of the `r` value.
	Since these bits can be tampered with, an attacker could in some cases deceive the algorithm by leading it to a wrong public key.
Recommendation	In order to verify the recovered key, Clef needs to:
	1. Verify that the key can be used to verify the signature passed in the request. This is step 1.6.2 of the SEC algorithm which is not implemented by Clef.
	2. Match the recovered public key against an Ethereum address or another authentication mechanism.
	To protect against these attacks, the API of Clef needs to be changed to accept an extra "authentication" argument.
Client Response	The ECRecover method was removed from Clef's API: github.com/holiman/go-ethereum/commit/cf3bf1724e58cc85ec87cb39a0aee0cb246c472e

---

[10]SEC 1: Elliptic Curve Cryptography version 2.0

Finding	**Outdated Dependencies**
Risk	**Low** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-EF-Clef-011
Category	Patching

Location	• `signer/rules/deps/bignumber.js` found at
	  – https://github.com/holiman/go-ethereum/blob/clefchanges_2/signer/rules/deps
	  – https://github.com/ethereum/go-ethereum/blob/master/signer/rules/deps
	• `vendor/vendor.json` found at
	  – https://github.com/holiman/go-ethereum/blob/clefchanges_2/vendor
	  – https://github.com/ethereum/go-ethereum/blob/master/vendor

Impact	Outdated dependencies may expose the application to publicly discovered vulnerabilities.
Description	Many of the largest breaches to date have relied on exploiting known vulnerabilities in outdated components. The Clef and Go-ethereum code repositories draw from many outdated components, albeit without currently well-known vulnerabilities. The risk is proportional to component functional and data sensitivity, development activity and quality, popularity, and length of time between project dependency updates. The OWASP project lists this risk on the Ten Most Critical Web Application Security Risks[11] due to the widespread prevalence of this issue. The `signer/rules/rules.go` code utilizes `signer/rules/deps/bindata.go` to essentially load the 'bignumber.js' library for arbitrary-precision decimal and non-decimal arithmetic. The source of this library is `signer/rules/deps/bignumber.js` which is version 2.0.3. The changelog for this project[12] indicates that this version was released in February 2015, and that the most recent version is 7.2.1. NCC Group is not aware of any publicly known vulnerabilities in this library. The `vendor/vendor.json` file lists approximately 154 Golang dependencies with revision timestamps ranging from early 2015 to August 2018. The majority of these are out of date and can be updated. For example, there seven components sourced from a UPnP client library for Go repository at `https://github.com/huin/goupnp` with a commit hash of `679507af18f3c7ba2bcc7905392ce23e148661c3` made on December 2016 which is 11 commits out of date.
Recommendation	Update the project dependencies to the most recent and stable versions recommended for production deployment. Include a regular review of dependency freshness as part of the development process.

[11] https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
[12] https://github.com/MikeMcl/bignumber.js/blob/master/CHANGELOG.md

Finding	**Rules Dangerously Rely On Time And State**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-EF-Clef-012
Category	Configuration
Location	• cmd/clef/rules.md
Impact	Attacks exist to alter Clef's state and access to time. If successfully mounted, they would allow an attacker to either revert the state used by Clef's rule or alter the time as seen and used by Clef. This could ultimately allow an attacker that has access to the Clef interface to remove some limitations imposed by its rules.
Description	Clef allow users to write rules (in javascript) in order to automate handling of the requests to Clef. Several examples are given in the documentation,[13] the first one being a time rule limiting how much ether can be sent over a window of 1 week. For this, the javascript run-time environment relies on the time given by the system:

```
var windowstart = new Date().getTime() - window;
```

There exist different ways for attackers to affect the time of the device running Clef without being root on the system:

1. If the `CAP_SYS_TIME` capability[14] is set on the `date` program, any user can change the time.
2. If the attacker has a privileged man-in-the-middle position in the network, she could attack the NTP protocol[15] to alter the device's time.

Furthermore, to keep a state in between executions of the rules, Clef keeps an encrypted key-value storage (`jsStorage`). Particular attacks might allow an attacker to alter this state and remove some limitations (for example if a boolean has ben set to prevent further transactions, reverting the state would allow transactions to flow again):

1. If the attacker has a physical access to the machine, she could re-set it to a previous snapshot.
2. If the attacker has file permissions to the `jsStorage`, she could record changes and revert the file to a previous point in time.

These attacks could allow an attacker to prevent certain rules from working correctly, or worse, to drain wallets if the attacker has direct access to Clef's API.

Recommendation	This finding highlights the fact that the security of the system running Clef is of utmost importance. Different type of users should be given different recommendations, depending on how much is at stake. Different threat models could be written, documenting what Clef protects against and does not protect against. Ultimately, it is hard to defend against these types of highly targeted attacks from powerful adversaries, and they should be out of the threat model of Clef.

---

[13] cmd/clef/rules.md
[14] http://man7.org/linux/man-pages/man7/capabilities.7.html
[15] https://www.cs.bu.edu/~goldbe/NTPattack.html

Finding	**Denial of Service Through Malformed Import Key**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-EF-Clef-013
Category	Data Validation
Location	• accounts/keystore/keystore_passphrase.go
Impact	A malicious attacker with access to the API or a privileged man-in-the-middle position could craft malicious import requests or tamper with them in order to crash the application or alter the keys imported without alerting the user.
Description	Clef's API exposes an "Import" method allowing requests to import already existing accounts. This import method accepts an encrypted key as argument, which must be formatted under specific formats. Two formats are accepted by Clef: a version 1 and a version 3, which utilize different encryption methods. Most of the code following the import of the key systematically assumes that the argument passed by the request is trusted, which might be in practice since rules cannot be written to handle this method: the user must manually accept an import request. The following code paths all have issues:

Importing Private Keys. The Import flow ends up calling the `crypto.ToECDSAUnsafe()` method which, as indicated, "blindly converts a binary blob to a private key. It should almost never be used unless you are sure the input is valid and want to avoid hitting errors due to bad origin encoding (0 prefixes cut off)."

JSON Parsing. Several fields from the passed JSON object are retrieved without previously checking if they exist. The getKDFKey() function used to retrieve KDF parameters from the request does not expect an empty map as `cryptoJSON.KDFParams` and will crash if given one. Additionally, it expects integers as fields for the KDF object, even when given strings.

KDF parameters. A denial of service can be obtained by providing absurdly large parameters for the Scrypt Key Derivation Function, which will force the program to compute an interminable cryptographic operation.

Authenticated Encryption. Before attempting the decryption of the imported key, the keystore will verify the integrity of the ciphertext in order to detect any tampering from man-in-the-middle attackers. This integrity check does not include the `IV` and is not done in constant time. This could allow an attacker to tamper with the `IV`, making the user decrypt the wrong private key, without any alert given by Clef (even though the "address" recovered is different from the "address" field passed as argument in the request).

Decryption. Version 1 of the Importer will use AES-CBC to decrypt the received key, in particular the low-level `CryptBlocks` function which takes a multiple of the blocksize as argument. If not, the function will panic, as can be seen in .[16]

Reproduction Steps	Run the following in the terminal with a Clef process exposing an RPC interface on `local-host:8550` and observe that the Clef application crashes.

```
curl -i -H "Content-Type: application/json" -X POST --data '{"jsonrpc":"2.0","me
➜  thod":"account_import","params":[{"version":"1","address":"string","id":"str
➜  ing","crypto":{"cipher":"","ciphertext":"","cipherparams":{"iv":""},"kdf":""
➜  ,"kdfparams":{},"mac":""}}],"id":67}' http://localhost:8550/
```

[16]crypto/cipher/cbc.go

The following payloads also crash the application:

```
{"version":"1","address":"string","id":"string","crypto":{"cipher":"","ciphertex
↪   t":"","cipherparams":{"iv":""},"kdf":"","kdfparams":{"salt":"","dklen":"","n
↪   ":"","r":"","p":"","c":"","prf":""},"mac":""}}
{"version":"1","address":"string","id":"string","crypto":{"cipher":"","ciphertex
↪   t":"01","cipherparams":{"iv":""},"kdf":"pbkdf2","kdfparams":{"salt":"","dkle
↪   n":5,"n":5,"r":5,"p":5,"c":5,"prf":"hmac-sha256"},"mac":"32f2f344a0bdf0434df
↪   8d3c3fd2afd043c1a26b969bb7c448abd67a4af27ae03"}}
```

Recommendation   Thoroughly document the fact that the Import API **MUST** be used with trusted and verified inputs. In addition, address the issues underlined in this finding.

Furthermore, consider removing the Import method from the API of Clef.

Finding	**Encrypted KeyStore Integrity Check Is Incomplete**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-EF-Clef-014
Category	Cryptography
Component	keystore
Location	• accounts/keystore/keystore_passphrase.go
Impact	An attacker can tamper with a wallet backup without alerting the user, who would not realize the assault until trying to use the wallet key.
Description	The keystore package of Go-Ethereum has an exported `EncryptKey()` method capable of storing wallets in an encrypted form. The encryption uses a key derived from a passphrase known by the user. As a mean of integrity check, to ensure that a backup of an encrypted key has not been tampered with, the keystore computes a message authentication code (MAC) over the ciphertext as can be seen below:

```
// EncryptKey encrypts a key using the specified scrypt parameters into a json
// blob that can be decrypted later on.
func EncryptKey(key *Key, auth string, scryptN, scryptP int) ([]byte, error) {
        // ...
        derivedKey, err := scrypt.Key(authArray, salt, scryptN, scryptR, scrypt
➜  P, scryptDKLen)
        // ...
        iv := make([]byte, aes.BlockSize) // 16
        if _, err := io.ReadFull(rand.Reader, iv); err != nil {
                panic("reading from crypto/rand failed: " + err.Error())
        }
        cipherText, err := aesCTRXOR(encryptKey, keyBytes, iv)
        // ...
        mac := crypto.Keccak256(derivedKey[16:32], cipherText)
```

This integrity check does not include the `iv`, important piece of the encryption/decryption process, allowing attackers to tamper with it without having to modify the ciphertext. Since the content encrypted is of high entropy, no strong attacks can be performed.

Recommendation	Make use of an authenticated cipher like AES-GCM which compiles encryption with integrity check as a single algorithm.

Finding	**UI Mixes Extraneous and Approval-Specific Data**
Risk	Informational Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-EF-Clef-006
Category	Data Validation
Location	`signer/core/cliui.go`
Impact	An attacker could phish a user through the display of attacker controlled information in the Clef UI.
Description	When Clef receives a request through its exposed API, metadata is displayed to the user in charge of handling it. This metadata includes a variety of fields unrelated to what is being signed like IP address, user-agent, origin, etc. There are 6 calls to `showMetadata()` within `signer/core/cliui.go` that drive this functionality. Some of these fields can be trivially tampered with and might provide a false understanding as users could rely too heavily on them instead of the important fields.

The following 'malicious' request (with the redacted JSON taken from Go code example comments) will be accepted by Clef.

```
curl http://localhost:8550/    \
  -i -H "Content-Type: application/json"    \
  -X POST --data '{...}'    \
  -A "indicates INVALID CHECKSUM IS EXPECTED"    \
  -H "Origin: NCC Group requires IMMEDIATE APPROVAL per direction of J Smith"
```

Clef will present the following information to the user.

```
--------- Transaction request-------------
to:     0x07a565b7ed7d7a678680a4c162885bedbb695fe0

WARNING: Invalid checksum on to-address!

from:     0x82A2A876D39022B3019932D30Cd9c97ad5616813 [chksum ok]
value:    16 wei
gas:      0x333 (819)
gasprice: 291 wei
nonce:    0x0 (0)
data:     0x4401a6e40000000000000000000000000000000000000000000000000000000000...012

Transaction validation:
  * WARNING : Invalid checksum on to-address
  * Info : safeSend(address: 0x0000000000000000000000000000000000000012)

Request context:
        127.0.0.1:40802 -> HTTP/1.1 -> localhost:8550

        User-Agent: indicates INVALID CHECKSUM IS EXPECTED
        Origin: NCC Group requires IMMEDIATE APPROVAL per direction of J Smith
-----------------------------------------
Approve? [y/N]:
>
```

As currently presented, the metadata provides little benefit to legitimate requests but may facilitate illegitimate requests. A naive user may consider the extraneous request data as

superseding the true warning above and mistakenly approve this transaction.

Recommendation	Do not present request metadata alongside approval-specific data without clear delineation and warnings. Either clearly label the categories presented and warn that request data cannot be relied upon, or simply remove all request data.
Client Response	An additional message was added before displaying metadata provided by the external caller of the API: github.com/holiman/go-ethereum/commit/c6d7644e5a5bd0fe23c7f060a390112 115515cab

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

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 mathematical protections for 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.

Ethereum Foundation / NCC Group Confidential

Appendix B: Project Contacts

The team from NCC Group has the following primary members:

- David Wong — Consultant
  david.wong@nccgroup.trust

- Eric Schorn — Consultant
  Eric.Schorn@nccgroup.trust

The team from Ethereum Foundation has the following primary member:

- Martin Swende — Ethereum Foundation
  martin.swende@ethereum.org