TrueSec

Document type	Document ID	Version	Date
Report		1.0	April 25, 2017
Created by	**Reviewed by**	**Classification**	
Philip Åkesson	Niclas Adlertz		

Go Ethereum Security Review

Ethereum Foundation (Stiftung Ethereum)

Andreas Hallberg, andreas.hallberg@truesec.se
Philip Åkesson, philip.akesson@truesec.se

TrueSec SYD AB	Phone	Fax	E-mail	Web
Drottninggatan 38	08-10 00 10	08-10 00 77	info@truesec.se	www.truesec.com
SE-211 41 Malmö				
Org.nr. 556919-7311				

Executive Summary

TrueSec has during April 2017 performed a security review of the Go implementation of Ethereum. TrueSec found the code to be of high quality and developed with a security-focused mindset. No critical security vulnerabilities have been found. The most serious vulnerability is an unintentional default bypass of web browsers' same-origin policy when enabling the client's RPC HTTP endpoint. The other issues found do not present direct attack vectors in themselves, and the rest of the report mainly consists of general comments and recommendations.

Contents

1 Purpose and scope

TrueSec has during April 2017 at the request of the Ethereum Foundation performed a security review of Go Ethereum[1], the official Go implementation of Ethereum.

The following components were in primary scope:

- Peer-to-peer communication and networking, block downloading
- RPC interface, Javascript scripting engine
- Transaction and block processing, consensus rules implementation
- Ethereum Virtual Machine (resource exhaustion/Denial-of-Service)

Specifically not in scope:

- Light client implementation, light client specific syncing (still being implemented)
- Backend database, leveldb storage, trie implementation
- Account handling, crypto, handling of private keys and wallets
- Swarm (not in production yet)

2 Document revisions

1.0	2017-04-25	Philip Åkesson	First version of the document

3 Methods

The review has mainly been done by reading the code and forming an understanding of how the application works. More specifically, TrueSec has:

- Tried to identify possible bottlenecks that can lead to Denial-of-Service (DoS)
- Verified that data shared between threads is properly protected
- Followed the data flow from external inputs to see if invalid input somehow can lead to erroneous behavior
- Performed fuzzing of the RLP serialization format (and the EVM bytecode interpreter, with the existing go-fuzz entry point)
- Automatically searched for race conditions using Go's built-in "–race" build flag

Dynamic analysis has been performed by running a private Ethereum network with two `geth` nodes and one `bootnode`, and by writing new unit tests as well as modifying existing tests.

[1] `https://ethereum.github.io/go-ethereum/`

4 Results

4.1 Peer-to-peer (p2p) and networking

TrueSec has reviewed the p2p and networking code, focusing on:

- Secure channel establishment - handshake and establishment of shared secrets
- Secure channel properties - confidentiality and integrity
- Message serialization
- Node discovery
- Protection against Denial-of-Service: timeouts and message size limits

TrueSec has also fuzzed the RLP decoding using go-fuzz[2] without finding any crashes.

4.1.1 Known issues

Although the shared secrets are properly established via the "encryption handshake", the channel lacks confidentiality due to a so called "two-time-pad"-flaw in the implementation of the symmetric encryption. This is an already known issue (see https://github.com/ethereum/devp2p/issues/32 and https://github.com/ethereum/go-ethereum/issues/1315). Since the channel today only transports public blockchain data, the issue has intentionally been left unresolved.

Another known issue is the lack of replay protection on the secure channel level (a defunct time-based replay protection mechanism was mentioned in conversation with the Ethereum developers). TrueSec recommends that the next version of the protocol implement replay protection using message numbers.

4.1.2 Unnecessarily large memory allocations

In rlpx.go, TrueSec found two end user controlled memory allocations that are unnecessarily large. TrueSec has not found a way to exploit them in a Denial-of-Service scenario, but recommends that they be validated more aggressively.

When reading a protocol message, 16.8MB can be allocated:

```
func (rw *rlpxFrameRW) ReadMsg() (msg Msg, err error) {
        ...
        fsize := readInt24(headbuf)
        // ignore protocol type for now

        // read the frame content
        var rsize = fsize // frame size rounded up to 16 byte boundary
        if padding := fsize % 16; padding > 0 {
                rsize += 16 - padding
        }
        // TRUESEC: user-controlled allocation of 16.8MB:
        framebuf := make([]byte, rsize)
        ...
}
```

Since the maximum message size of the ethereum protocol is 10MB, TrueSec recommends that the same size limit is applied at this level before consuming the rest of the message.

[2] https://github.com/dvyukov/go-fuzz/

During the encryption handshake, it is possible to allocate 65KB for the handshake message:

```
func readHandshakeMsg(msg plainDecoder, plainSize int,
                      prv *ecdsa.PrivateKey, r io.Reader) ([]byte, error) {
        ...
        // Could be EIP-8 format, try that.
        prefix := buf[:2]
        size := binary.BigEndian.Uint16(prefix)
        if size < uint16(plainSize) {
                return buf, fmt.Errorf("size underflow, need at least ...
        }
        // TRUESEC: user-controlled allocation of 65KB:
        buf = append(buf, make([]byte, size-uint16(plainSize)+2)...)
        ...
}
```

Unless handshake message data is expected to actually contain 65KB of data TrueSec recommends that the size is more aggressively checked before consuming the rest of the message.

4.2 Transaction and block processing

TrueSec reviewed the transaction and block downloading and processing parts, focusing on:

- Denial-of-Service by memory allocation, goroutine leaks and I/O operations
- Synchronization problems

4.2.1 Divide-by-zero risk

In Go, dividing by zero results in a panic. In `downloader.go` the method `qosReduceConfidence` relies on the caller having ensured that `peers` is not zero before dividing:

```
func (d *Downloader) qosReduceConfidence() {
        peers := uint64(d.peers.Len())
        ...
        // TRUESEC: no zero-check of peers here
        conf := atomic.LoadUint64(&d.rttConfidence) * (peers - 1) / peers
        ...
}
```

TrueSec has not found a way of exploiting this to crash the node, but relying on the caller to ensure that `d.peers.Len()` is not zero is fragile. TrueSec recommends that all non-constant divisors are zero-checked immediately before dividing.

4.2.2 Code complexity

TrueSec found the transaction and block processing code to be more complex and harder to read than the code in the other areas in this audit. Methods tend to be larger than usual, with methods in `fetcher.go`, `downloader.go` and `blockchain.go` near or above the 200 LOC mark. Synchronization is achieved with a sometimes rather involved combination of mutexes and channel messaging. As an example, the `Downloader` struct definition (`downloader.go`) needs 60 LOC with 3 mutexes and 11 channels.

Code that is hard to read and understand is fertile ground for security issues. The `eth` package in particular contains several large methods, structs and interfaces, as well as extensive use of mutexes and channels. TrueSec recommends that effort is spent refactoring and simplifying the code to prevent security problems in the future.

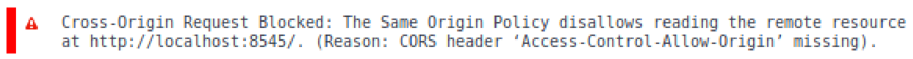
4.3 IPC and RPC interfaces

TrueSec reviewed the IPC and RPC (HTTP and WS) interfaces, with a focus on potential access control issues and ways to escalate privileges from public APIs to private APIs (admin, debug et cetera).

4.3.1 CORS: All origins allowed by default in the HTTP RPC

The HTTP RPC interface can be enabled by starting `geth` with `--rpc`. This will start a web server listening for HTTP requests on port 8545, which can be accessed by anyone with network access. Because of the potential for exposing this by mistake (for example by connecting to an untrusted network), only public APIs are available on the HTTP RPC interface by default.

The same-origin policy and default cross-origin resource sharing (CORS) settings restrict web browser access and limits the possibilities of attacking the RPC API via, for example, cross-site scripting (XSS). The allowed origins can for example be configured with `--rpccorsdomain "domain"`, as a comma-separated list `--rpccorsdomain "domain1,domain2"`, or the special case `--rpccorsdomain "*"` which essentially allows all domains full access from standard web browsers. If not configured, the CORS headers will not be set - and the browser will not allow cross-origin requests to be made. See Figure 1.



```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource
at http://localhost:8545/. (Reason: CORS header 'Access-Control-Allow-Origin' missing).
```

Figure 1: Firefox blocking a cross-origin request due to missing CORS headers

However, this was broken in commit `5e29f4b` [3] (from Apr 12, 2017) - resulting in the RPC being accessible through web browsers in a way that was not intended, essentially bypassing the same-origin policy.

The CORS configuration for the HTTP RPC was changed to handle the allowed origins in a string array - instead of passing it around internally as a single comma-separated string.

Previously, the comma-separated string was split into an array just before instantiating the `cors` middleware (see Listing 1). With the default value (when the user has not explicitly configured anything, for example using `--rpccorsdomain`) being an empty string, this resulted in a string array containing an empty string.

After commit `5e29f4b`, the default value is instead an empty array which is passed all the way through to the `cors` middleware in `newCorsHandler` (see Listing 2).

The `cors` middleware then checks the length of the allowed origins array (see Listing 3). If the length is zero, as is the case for an empty array, the cors middleware [4] will fall back to its own default value and allow all origins.

The issue can be shown by running `geth --rpc`, without specifying any allowed origins, and checking for the CORS headers with an `OPTION` request before (Listing 4) and after (Listing 5) commit `5e29f4b`. Note the value of `Access-Control-Allow-Origin` in the second output.

Note that this would have been the case even before the changes, had it not been for the string splitting resulting in `cors` not interpreting the input value (an array containing an empty string) as empty.

The issue can be exploited from a browser with the following JavaScript code, executing from any domain (even a local filesystem, which will in most cases result in an invalid or `null` Origin):

[3] https://github.com/ethereum/go-ethereum/commit/5e29f4be935ff227bbf07a0c6e80e8809f5e0202
[4] `https://github.com/rs/cors`

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://localhost:8545", true);
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function() {
  if (xhr.readyState == XMLHttpRequest.DONE && xhr.status == 200) {
    console.log("Modules: " + xhr.responseText);
  }
}
xhr.send('{"jsonrpc":"2.0","method":"rpc_modules","params":[],"id":67}')
```

TrueSec recommends that the default configuration for CORS is explicitly set as restrictive as possible (eg. allowed origin set to `localhost`, or even no CORS headers at all), instead of relying on an external dependency to choose a sane (and secure) default configuration.

```
func newCorsHandler(srv *Server, corsString string) http.Handler {
        var allowedOrigins []string
        for _, domain := range strings.Split(corsString, ",") {
                allowedOrigins = append(allowedOrigins, strings.TrimSpace(domain))
        }
        c := cors.New(cors.Options{
                AllowedOrigins: allowedOrigins,
                AllowedMethods: []string{"POST", "GET"},
                MaxAge:         600,
                AllowedHeaders: []string{"*"},
        })
        return c.Handler(srv)
}
```

Listing 1: rpc/http.go, before commit `5e29f4be935ff227bbf07a0c6e80e8809f5e0202`

```
func newCorsHandler(srv *Server, allowedOrigins []string) http.Handler {
        c := cors.New(cors.Options{
                AllowedOrigins: allowedOrigins,
                AllowedMethods: []string{"POST", "GET"},
                MaxAge:         600,
                AllowedHeaders: []string{"*"},
        })
        return c.Handler(srv)
}
```

Listing 2: rpc/http.go, after commit `5e29f4be935ff227bbf07a0c6e80e8809f5e0202`

```go
	// Allowed Origins
	if len(options.AllowedOrigins) == 0 {
		// Default is all origins
		c.allowedOriginsAll = true
	}
```

Listing 3: vendor/github.com/rs/cors/cors.go

```
$ curl -i -X OPTIONS
   -H "Access-Control-Request-Method: POST"
   -H "Access-Control-Request-Headers: content-type"
   -H "Origin: foobar" http://localhost:8545

HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Date: Tue, 25 Apr 2017 08:49:10 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Listing 4: CORS headers before commit `5e29f4b`

```
$ curl -i -X OPTIONS
   -H "Access-Control-Request-Method: POST"
   -H "Access-Control-Request-Headers: content-type"
   -H "Origin: foobar" http://localhost:8545
HTTP/1.1 200 OK
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Methods: POST
Access-Control-Allow-Origin: foobar
Access-Control-Max-Age: 600
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Date: Tue, 25 Apr 2017 08:47:24 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

Listing 5: CORS headers after commit `5e29f4b`

4.4 JavaScript Engine and API

The JavaScript engine `otto`[5] is included in Go Ethereum as a CLI scripting interface, a console REPL to the IPC/RPC interfaces, and as a part of the private debug API. Given the limited exposure of this code, this was considered a lower priority during the review.

4.4.1 Weak seed in pseudo-random number generation

When initializing the pseudo random number generator used in the `jsre`, the seed falls back to the current UNIX time if `crypto/rand` (which returns cryptographically secure pseudorandom numbers) fails. See listing 6. The seed is then used to initialize an instance of `math/rand`.

This PRNG is not used for anything sensitive internally, and obviously should not be used as a cryptographically secure RNG, but since it can be used by end-users when running scripts from the command-line interface it might be safer to fail instead of potentially providing weak seeds. Getting errors from `crypto/rand` is likely indicative of other issues as well.

Even with decent seeds, it should also be made very clear in the documentation that this PRNG is not cryptographically secure.

```go
// randomSource returns a pseudo random value generator.
func randomSource() *rand.Rand {
        bytes := make([]byte, 8)
        seed := time.Now().UnixNano()
        if _, err := crand.Read(bytes); err == nil {
                seed = int64(binary.LittleEndian.Uint64(bytes))
        }

        src := rand.NewSource(seed)
        return rand.New(src)
}
```

Listing 6: `internal/jsre/jsre.go`

[5] `https://github.com/robertkrimen/otto`

4.5 EVM implementation

TrueSec reviewed the Ethereum Virtual Machine (EVM) with focus on Denial-of-Service by abusing memory allocation and I/O usage. There was an existing go-fuzz entry point to the EVM interpreter (`runtime/fuzz.go`), which appears to have been already used with success. TrueSec verified its functionality, but found no issues during fuzzing.

4.5.1 Cheap memory consumption by abusing the intPool

For performance reasons, big integers used during the EVM's execution are pooled in the `intPool` (`intpool.go`). The pool has no size limit, leading to an unintentionally cheap way of consuming memory using specific opcode combinations.

For example, the contract code

```
0 JUMPDEST     // 1 gas
1 COINBASE     // 2 gas
2 ORIGIN       // 2 gas
3 EQ           // 3 gas, puts 20 + 20 bytes on the intpool
4 JUMP         // 8 gas, puts 4-8 bytes on the intpool
```

would (if the block gaslimit permitted it, see below) allocate 10GB on the intPool for 3.33e9 gas (about 3300 USD at the time of writing). The EVM's intended gas cost for 10GB is 1.95e14 gas (about 195 million USD).

A DoS-attack by provoking an out-of-memory panic via `intPool` is prevented by the consensus rules, restricting the gaslimit growth to 1/1024 per block and having the gaslimit target at 4.7e6 gas. TrueSec still recommends that the size of the `intpool` be restricted, should an attacker discover a more efficient way of populating the intPool or should the gaslimit target increase drastically.

4.5.2 Fragile protection of negative-valued transactions in mined blocks

Transfer of Ether between accounts are done by the method `Transfer` in `core/evm.go`:

```
func Transfer(db vm.StateDB, sender, recipient common.Address, amount *big.Int) {
        db.SubBalance(sender, amount)
        db.AddBalance(recipient, amount)
}
```

The input `amount` is a pointer to a signed type, potentially having a negative referenced value. A negative amount would move Ether from the recipient to the sender, effectively letting the sender steal Ether from the "recipient".

When receiving an unmined transaction, the transaction's value is validated to be positive. See `tx_pool.go`, `validateTx()`:

```
if tx.Value().Sign() < 0 {
                return ErrNegativeValue
}
```

During block processing, there is no such explicit validation; transactions with negative values are prevented only implicitly by the p2p serialization format (RLP) which can not decode negative values. Considering an evil miner stealing Ether by issuing blocks with negative-valued transactions, relying on the particular serialization format to provide this protection seems unnecessarily fragile.

TrueSec recommends that explicit validation of the transaction's value be done also during block processing. One could also consider enforcing the unsignedness of a transaction's amount by using an unsigned type.

4.6 Miscellaneous

4.6.1 Race condition in mining code

TrueSec used the ”–race” build flag to find race conditions using Go's built-in race detection feature. A race condition was discovered in `ethash/ethash.go` concerning the timestamping of ethash-datasets used when mining:

```
func (ethash *Ethash) dataset(block uint64) []uint32 {
        epoch := block / epochLength

        // If we have a PoW for that epoch, use that
        ethash.lock.Lock()
        ...
        current.used = time.Now() // TRUESEC: race
        ethash.lock.Unlock()

        // Wait for generation finish, bump the timestamp and finalize the cache
        current.generate(ethash.dagdir, ethash.dagsondisk, ethash.tester)

        current.lock.Lock()
        current.used = time.Now()
        current.lock.Unlock()
        ...
}
```

To remove the race condition, protect the first setting of `current.used` with the `current.lock` mutex.

TrueSec has not investigated whether the race condition has in fact had any effect on the node's mining.

4.6.2 Many third-party dependencies

Go Ethereum depends on seventy one third-party packages (listed using `govendor list +vend`).

Since each dependency can introduce new attack vectors, and requires time and effort to monitor for security vulnerabilities, TrueSec always recommends that the number of third-party dependencies be kept at a minimum.

Seventy one dependencies seems like a lot to handle for any project. TrueSec recommends that the Ethereum developers investigate whether all dependencies are in fact needed, or if some of them can be replaced by own code.