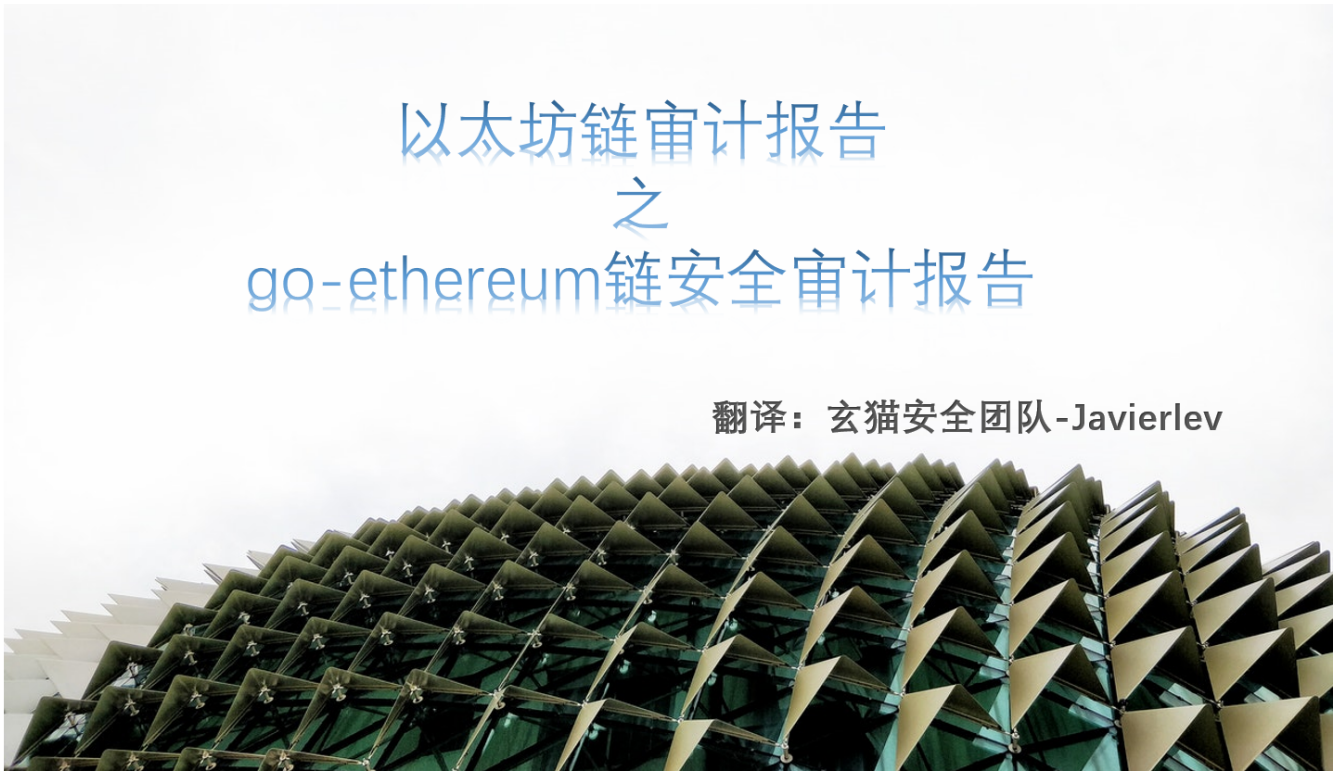
[翻译]以太坊链审计报告之go-ethereum链安全审计



翻译：玄猫安全团队-Javierlev

原文：`https://github.com/ethereum/go-ethereum/blob/master/docs/audits/2017-04-25_Geth-audit_Truesec.pdf`

近期，以太坊`go-ethereum`公开了两份审计报告，玄猫安全团队第一时间对其进行了翻译工作。此为第一篇《Go Ethereum Security Review》即`2017-04-25_Geth-audit_Truesec`，此审计报告完成时间为2017年4月25日。

如果您正在使用的是较旧版本的`go-ethereum`，强烈建议升级至最新版本，以免遭受不必要的损失。

1.1. 概述

`TrueSec`在2017年4月对以太坊的GO语言实现进行了代码审计。审计结果表明代码质量是比较高的，且开发者具备一定的安全意识。在审计过程中没有发现严重的安全漏洞。最严重的一个漏洞是当客户端的RPC HTTP开启时，web浏览器同源策略的绕过。其他发现的问题并没有直接的攻击向量可供利用，报告的其他部分为通用的评论和建议。

1.2. 目录

1.2.1. P2P和网络

- 已知问题
- 内存分配过大

1.2.2. 交易和区块处理

- 零除风险
- 代码复杂性

1.2.3. IPC和RPC接口

- CORS:在HTTP RPC中默认允许所有域

1.2.4. Javascript引擎和API

- 伪随机数生成器的弱随机种子

1.2.5. EVM实现

- 滥用`intPool`导致廉价的内存消耗
- 在挖矿区块中脆弱的负值保护

1.2.6. 杂项

- 在挖矿代码中的条件竞争
- 许多第三方依赖

1.3. 结果细节

1.3.1. P2P和网络

`TrueSec`对`p2p`和网络部分的代码进行了审计，主要关注：

- 安全的通道实现 - 握手和共享`secrets`的实现
- 安全的通道属性 - 保密性和完整性
- 消息的序列化
- 节点发现
- 对于DOS的防范：超时和消息大小限制

`TrueSec`还通过go-fuzz对RLP解码进行fuzz，没有发现节点崩溃的现象。

1.3.1.1. 已知问题

虽然`共享secrets`在`encryption handshake`中实现得比较好，但是由于在对称加密算法中的`two-time-pad`缺陷，使得通道缺乏保密性。这个是已知的问题。(详情参考`https://github.com/ethereum/devp2p/issues/32`和`https://github.com/ethereum/go-ethereum/issues/1315`)。由于现在通道只传输公开的区块链数据，这个问题暂时不必解决。

另外一个一直存在的问题是在安全的通道等级(在以太坊开发者讨论中提到过一个默认的基于时间的重放保护机制)中缺乏`重放保护`。`TrueSec`建议协议的下一个版本通过控制消息数量来实现`重放保护`。

1.3.1.2. 内存分配过大

在`rlpx.go`，`TrueSec`发现两个用户可控的，过大的内存分配。`TrueSec`没有发现可以利用的DOS情景，但是建议恰当地对其进行验证。

当读取协议消息时，`16.8MB`大小的内存可以被分配:

```
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)
    ...
}
```

由于以太坊协议中，对消息大小的最大值定义为`10MB`,`TrueSec`推荐内存分配也定义为相同大小。

在`encryption handshake`过程中，可以给握手信息分配`65KB`大小内存。

```
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)...)
    ...
}
```

除非握手消息确实包含`65KB`大小的数据，`TrueSec`建议对握手消息的大小作限制。

1.3.2. 交易和区块处理

`TrueSec`对交易和区块下载，区块处理的部分进行了代码审计，主要关注:

- 由内存分配，`gorountine`泄露和`IO操作`导致的拒绝服务
- 同步问题

1.3.2.1. 零除风险

在Go中，除以零会导致一个`panic`。在`downloader.go`的`qosReduceConfidence`方法中，是否出现零除取决于调用者正确调用:

```
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`没有发现可以导致节点崩溃的利用方式，但是仅仅依赖调用者来保证`d.peers.Len()`不为零是不安全的。`TrueSec`建议所有非常数的被除数应该在进行除法之前进行检查。

1.3.2.2. 代码复杂性

`TrueSec`发现交易和区块处理的代码部分相对其他部分代码来说更加复杂，更难阅读和审计。这部分的方法相对更大，在`fetcher.go`,`downloader.go`和`blockchain.go`中有超过200行的代码。同步的实现有时候会结合互斥锁和通道消息。比如说，结构体`Downloader`定义需要60行代码，包含3个互斥锁和11个通道。

难以阅读和理解的代码是滋生安全问题的肥沃土壤。特别是`eth`包中存在一些代码量大的方法，结构体，接口与扩展的互斥锁和通道。`TrueSec`建议花一些功夫重构和简化代码，来防止未来安全问题的发生。

1.3.3. IPC和RPC接口

`TrueSec`对IPC和RPC(HTTP和Websocket)接口进行了审计，关注于潜在的访问控制问题，从公共API提权到私有API(admin, debug等)的问题。

1.3.3.1. CORS:在默认的HTTP RPC里允许所有域

`HTTP RPC`接口可以通过`geth`的`--rpc`参数开启。这会启动一个web服务器，用于监听8545端口的HTTP请求，且任何人都可以对其进行访问。由于潜在暴露端口的可能性(比如连接到不可信的网络)，默认只有公共API允许HTTP RPC接口。

同源策略和默认的跨域资源共享(CORS)配置限制了web浏览器的访问，并且限制通过XSS攻击RPC API的可能性。`allowed origins`能够通过`--rpccorsdomain "domain"`来配置，也可以通过逗号分隔来配置多个域名`--rpccorsdomain "domain1,domain2"`，或者配置为`--rpccorsdomain "*"`,使得所有的域都可以通过标准web浏览器访问。如果没有进行配置，CORS头将不会被设置——并且浏览器不会允许跨域请求。如图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').
```

图1: 由于缺少CORS头，`Firefox`禁止跨域请求

但是，在[commit 5e29f4b]中(从2017年4月12日开始)——同源策略可以被绕过，RPC可以从web浏览器被访问。

HTTP RPC的CORS配置被改变为处理`allowed origins`的字符数组——而不是在内部作为一个单引号分隔的字符串传输。

在此之前，逗号分隔的字符串被分成一个数组，在实例化`cors`中间件之前(请见Listing 1)。with默认值(防用户没有显性配置任何设置时，如使用--rpccorsdomain)空字符串，这会导致一个字符数组包含一个空字符串。

在`commit 5e29f4b`之后，默认值是一个空的数组，这个数组传递给位于`newCorsHandler`的中间件`cors`(请见Listing 2)。

`cors`中间件随后检查`allowed origins`数组的长度(请见 Listing 3)。如果长度为0，在这里即代表空数组，cors中间件将会变成默认值并且允许所有域。

这个问题可以通过运行`geth -rpc`来复现，不需要指定任何`allowed origins`,并检查`commit 5e29f4b`前后带有`OPTION`请求的CORS头。第二个输出的`Access-Control-Allow-Origin`值得注意。

注意即使是改变之前，这里也是这样。如果不是因为字符串分割导致在`cors`没有解释输入值(一个数组包含一个空字符串)为空。

这个问题可以通过下面的`JavaScript`代码来利用，从任意域来执行(甚至可以是本地文件系统，即无效或者null origin)

```
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`建议将CORS的默认配置进行显性的限制，(如将`allowed origin`设置为`localhost`,或根本不设置CORS头)，而不是依赖外界来选择一个正常(安全)的默认设置

```
165 func newCorsHandler(srv *Server, corsString string) http.Handler {
166     var allowedOrigins []string
167     for _, domain := range strings.Split(corsString, ",") {
168         allowedOrigins = append(allowedOrigins, strings.TrimSpace(domain))
169     }
170     c := cors.New(cors.Options{
171         AllowedOrigins: allowedOrigins,
172         AllowedMethods: []string{"POST", "GET"},
173         MaxAge: 600,
174         AllowedHeaders: []string{"*"},
175     })
176     return c.Handler(srv)
177 }
```

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

```
164 func newCorsHandler(srv *Server, allowedOrigins []string) http.Handler {
165     c := cors.New(cors.Options{
166         AllowedOrigins: allowedOrigins,
167         AllowedMethods: []string{"POST", "GET"},
168         MaxAge: 600,
169         AllowedHeaders: []string{"*"},
170     })
171     return c.Handler(srv)
172 }
```

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

```
113     // Allowed Origins
114     if len(options.AllowedOrigins) == 0 {
115     // Default is all origins
116     c.allowedOriginsAll = true
117     }
```

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

1.3.4. JavaScript引擎和API

JavaScript引擎otto是Go Ethereum中的CLI脚本接口，一个IPC/RPC接口的终端交互解释器，也是私有`debug API`的一部分。考虑到其代码有限，在审计中优先级比较低。

1.3.4.1. 伪随机数生成的弱随机数种子

在 `jsre` 中对伪随机数生成器进行初始化的时候，如果 `crypto/rand`（`crypto/rand` 返回密码学安全地伪随机数)方法失败，随机数种子将会依赖于当时的UNIX时间。在listing 6中，这个弱随机数种子将会被用于初始化 `math/rand` 的实例。

这个 `PRNG` 没有被用于任何敏感信息，而且显然也不应该被用作于密码学安全的 `RNG`,但是由于用户可以通过命令行运行脚本来使用 `PRNG`，使其失败而不是制造出弱随机数种子显然是更安全的。从 `crypto/rand` 中得到错误意味着其他地方可能也存在问题。即使是得到了安全的随机数种子，在文档中也应该指出 `PRNG` 并不是密码学安全的。

```
84 // randomSource returns a pseudo random value generator.
85 func randomSource() *rand.Rand {
86     bytes := make([]byte, 8)
87     seed := time.Now().UnixNano() // 不是完全随机
88     if _, err := crand.Read(bytes); err == nil {
89         seed = int64(binary.LittleEndian.Uint64(bytes))
90     }
91
92     src := rand.NewSource(seed)
93     return rand.New(src)
94 }
```

Listing 6: internal/jsre/jsre.go

1.3.5. 以太坊虚拟机(EVM)的实现

`TrueSec` 对以太坊虚拟机(EVM)部分的代码进行了审计，主要关注由滥用内存分配和IO操作而引起的拒绝服务。EVM解释器(runtime/fuzz.go)存在一个 `go-fuzz` 的入口点，这个入口点成功地被使用。 `TrueSec` 确认了其功能性，但是在fuzzing过程中没有发现有影响的漏洞。

1.3.5.1. 滥用intPool导致的廉价的内存消耗

由于性能的原因，在EVM的执行过程中，使用大整数会进入整数池 `intPool` (intpool.go)。由于没有对整数池大小进行限制，使用特定的 `opcode` 组合，将导致意外出现廉价使用内存的情况。

```
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
```

比如说，合约代码将会消耗3.33e9单位的gas(在当时大约价值3300USD)，分配10G内存给 `intPool`。以太坊虚拟机中分配10GB内存的预期 `gas` 成本是1.95e14(大约195,000,000USD)

当 `intPool` 产生 `out of memory panic` 时，会导致拒绝服务攻击。但是共识算法对 `gaslimit` 进行了限制，能够阻止该拒绝服务攻击的发生。但是考虑到攻击者可能发现一种更有效的填充 `intPool` 的方式，或者 `gaslimit target` 增长过于迅速等， `TrueSec` 仍然推荐对 `intPool` 的大小进行限制。

1.3.5.2. 在挖矿区块中脆弱的负值保护

账户之间以太坊的转账是通过 `core/evm.go` 里的 `Transfer` 方法进行的。

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

输入`amount`是一个指向有符号类型的指针，可能存在负的引用值。一个负的`amount`将会把以太坊从收款方转移到转账方，使得转账方可以从收款方那里盗窃以太坊。

当接收到一个没有被打包的交易时，将会验证交易的值是否为正。如`tx_pool.go, validateTx()`:

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

但是在区块处理过程中却不存在这样显性的验证；存在负值的交易只是隐性地被`p2p`序列化格式(RLP)阻止，而RLP不能解码负值。假设一个邪恶的矿工为了非法获取以太坊，发布了具有负值交易的区块，这时依赖于特定的序列化格式来提供保护，似乎有些脆弱。`TrueSec`推荐在区块处理过程中也显性地检查交易的值。或者使用无符号类型来强制指定交易的值为正。

1.3.6. 杂项

1.3.6.1. 在挖矿代码中的条件竞争

`TrueSec`使用"-race"来构建标志位，并通过`Go`语言内置的条件竞争探测特性来寻找条件竞争。在`ethash/ethash.go`中发现了一个与在挖矿时使用的`ethash datasets`时间戳相关的条件竞争。

```
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()
...
}
```

为了去除条件竞争，通过使用`current.lock`互斥锁可以保护第一个`current.used`的设置。`TrueSec`没有研究条件竞争是否会对节点的挖矿造成影响。

1.3.6.2. 过多第三方依赖

`Go Ethereum`依赖于71个第三方包(通过`govendor list +vend`列举)

由于每个依赖都可能引入新的攻击向量，并且需要时间和精力来监控安全漏洞，`TrueSec`总是建议将第三方包的数量控制到最小。

71个依赖对任何一个项目来说都是比较多的。`TrueSec`推荐以太坊开发者调研是否所有的依赖都是真正需要的，或者说其中一些是否可以用代码来替代。

1.4. 附录

1.4.1. 声明

我们努力提供准确的翻译，可能有些部分不太准确，部分内容不太重要并没有进行翻译，如有需要请参见原文。

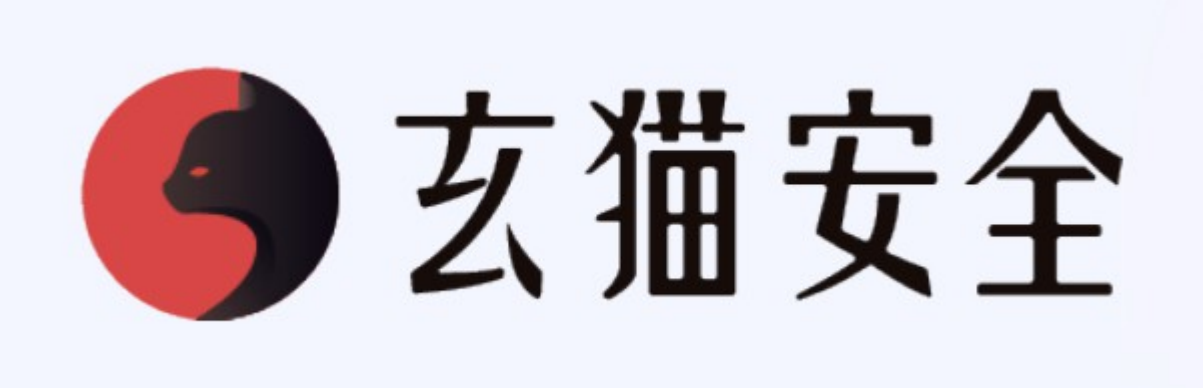
1.4.2. 原文地址

https://github.com/ethereum/go-ethereum/blob/master/docs/audits/2017-04-25_Geth-audit_Truesec.pdf

1.4.3. 参考链接

参考项目	URL地址
go ethereum	https://ethereum.github.io/go-ethereum/
go fuzz	https://github.com/dvyukov/go-fuzz/
commit 5e29f4	https://github.com/ethereum/go-ethereum/commit/5e29f4be935ff227bbf07a0c6e80e8809f5e0202
cors中间件	https://github.com/rs/cors
otto	https://github.com/robertkrimen/otto

关于玄猫安全



玄猫区块链安全实验室专注区块链安全领域，致力于提供区块链行业最专业的安全解决方案，团队成员来自于百度、阿里、360等国际顶尖安全团队，已为数十家交易所、电子钱包、智能合约等提供基础安全建设、渗透测试、漏洞挖掘、应急响应等安全服务。

玄猫安全实验室提供专业权威的智能合约审计服务、区块链专项应用评估、区块链平台安全评估等多项服务。

商务合作：Lyon.chen@xuanmao.org