Anyswap

智能合约安全审计报告

2020-07-20

1. 概要

慢雾安全团队于 2020 年 07 月 13 日，收到 Anyswap 团队对 Anyswap 系统安全审计的申请，根据项目特点慢雾安全团队制定如下审计方案。

慢雾安全团队将采用“白盒为主，黑灰为辅”的策略，以最贴近真实攻击的方式，对项目进行安全审计。

慢雾科技 DApp 项目测试方法：

黑盒测试	站在外部从攻击者角度进行安全测试。
灰盒测试	通过脚本工具对代码模块进行安全测试，观察内部运行状态，挖掘弱点。
白盒测试	基于项目的源代码，进行脆弱性分析和漏洞挖掘。

慢雾科技 DApp 漏洞风险等级：

严重漏洞	严重漏洞会对项目的安全造成重大影响，强烈建议修复严重漏洞。
高危漏洞	高危漏洞会影响项目的正常运行，强烈建议修复高危漏洞。
中危漏洞	中危漏洞会影响项目的运行，建议修复中危漏洞。
低危漏洞	低危漏洞可能在特定场景中会影响项目的业务操作，建议项目方自行评估和考虑这些问题是否需要修复。
弱点	理论上存在安全隐患，但工程上极难复现。
增强建议	编码或架构存在更好的实践方法。

2. 审计方法

慢雾安全团队智能合约安全审计流程包含两个步骤:

- 使用开源或内部自动化分析的工具对合约代码中常见的安全漏洞进行扫描和测试。
- 人工审计代码的安全问题，通过人工分析合约代码，发现代码中潜在的安全问题。

如下是合约代码审计过程中我们会重点审查的漏洞列表:

（其他未知安全漏洞不包含在本次审计责任范围）

- 重入攻击
- 重放攻击
- 重排攻击
- 短地址攻击
- 拒绝服务攻击
- 交易顺序依赖
- 条件竞争攻击
- 权限控制攻击
- 整数上溢/下溢攻击
- 时间戳依赖攻击
- Gas 使用，Gas 限制和循环
- 冗余的回调函数
- 不安全的接口使用
- 函数状态变量的显式可见性
- 逻辑缺陷
- 未声明的存储指针
- 算术精度误差
- tx.origin 身份验证
- 假充值漏洞
- 变量覆盖

3. 项目概述

3.1 项目描述

Anyswap 是一个去中心化的跨链交易协议，支持 BTC、ETH、USDT、FSN 等主流加密货币实时交易，自动价格，自动流动性以及 token 激励。

项目官网地址：https://anyswap.exchange/

项目测试网：FUSION 测试网

审计合约文件：

ANY 治理代币合约: https://github.com/anyswap/anyswap-token

commit: 937c687c78b80d4d554877b7254ac6a2166fc3ae

ANY 治理代币锁定合约: https://github.com/anyswap/ANYToken-locked

commit: 6e61beea7f95c25465291d7f79c0ce5e537f6dc5

Anyswap 交易合约: https://github.com/anyswap/anyswap-exchange

commit: 1c9cc0053b315535fc1c7f6cd6513888dd1a204d

项目方提供文件:

HowItWorks.docx:

MD5: 09d557ae2eb3971e102787830bc08b33

ANY 审计需求说明.docx:

MD5: 0bd0c67d98b503d541e3de0760194104

3.2 项目结构

anyswap-token:

```
.
├── AnyswapToken.sol
├── LICENSE
├── README.md
├── abi
│   └── AnyswapToken.json
├── bytecode
│   └── AnyswapToken.txt
├── contracts
│   ├── AnyswapToken.sol
│   └── Migrations.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_deploy_contracts.js
├── package-lock.json
├── package.json
└── truffle-config.js
```

ANYToken-locked:

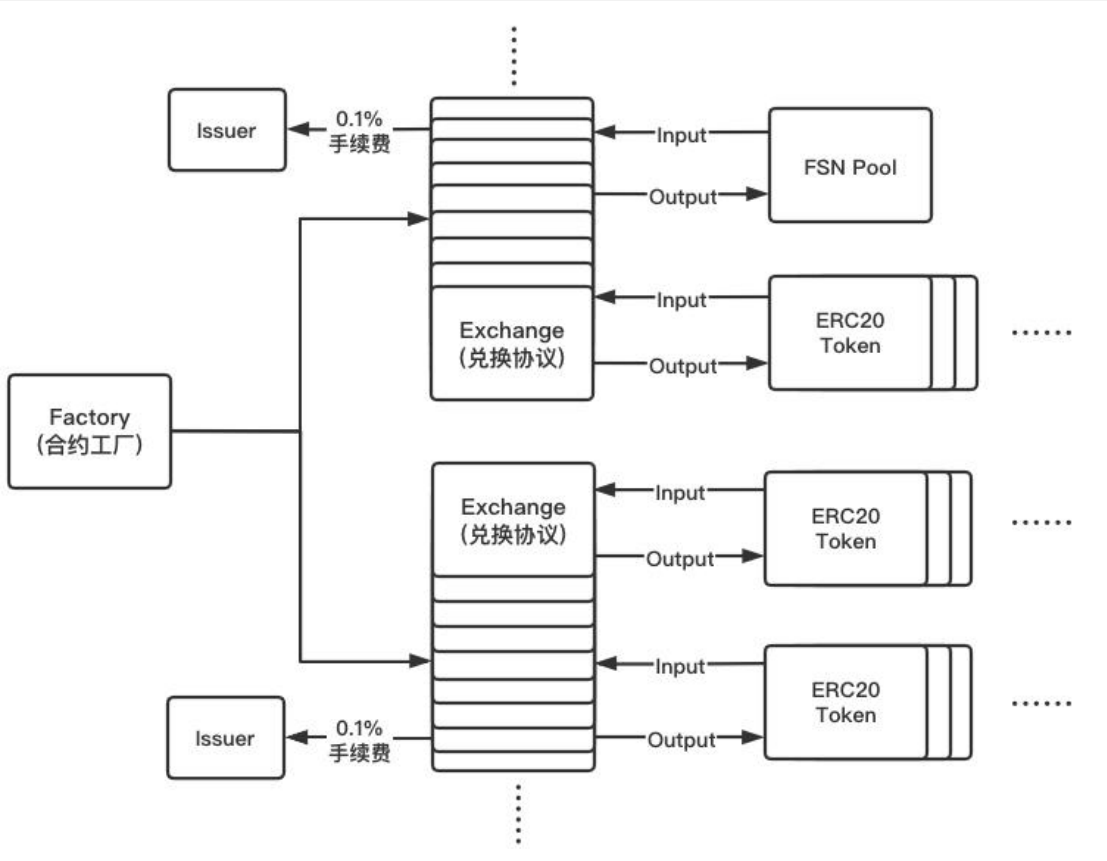
```
.
├── Distribute.sol
├── README.md
├── abi
│   └── Distribute.json
├── contracts
│   ├── Distribute.sol
│   └── Migrations.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_deploy_contracts.js
├── package-lock.json
├── package.json
└── truffle-config.js
```

```
anyswap-exchange:
.
├── LICENSE.md
├── README.md
├── abi
│   ├── uniswap_exchange.json
│   └── uniswap_factory.json
├── bytecode
│   ├── exchange.txt
│   └── factory.txt
├── contracts
│   ├── test_contracts
│   │   └── ERC20.vy
│   ├── uniswap_exchange.vy
│   └── uniswap_factory.vy
├── requirements.txt
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── constants.py
    └── exchange
        ├── test_ERC20.py
        ├── test_eth_to_token.py
        ├── test_factory.py
        ├── test_liquidity_pool.py
        ├── test_token_to_eth.py
        ├── test_token_to_exchange.py
        └── test_token_to_token.py
```

3.3 合约架构

Anyswap DApp 主要分为两个部分，分别为合约工厂与代币兑换部分。其中 uniswap_factory 合约负责实现为每个 ERC20 代币创建一个独立的交易合约。uniswap_exchange 合约负责实现提供流动性池进行代币兑换、手续费处理以及自定义资金池的功能。每个交易合约都会与一个 ERC20 Token 关联，并且带有一个 FSN 和这个 ERC20 Token 的流动性池，以实现 FSN 与 Token 的兑换功能以及 Token 与 Token 之间的兑换功能，兑换过程中产生的手续费有一部分存入流动性准备金中，另一部分被发送到

issuer 的钱包中。合约整体架构图如下：



4. 代码分析

4.1 主要合约文件及哈希

No	File Name	SHA-1 Hash
1	AnyswapToken.sol	5d3e5d61957cde64213aefa96c785eadc6b5d444
2	Distribute.sol	116b7f7b9867bb668c4f3d4aa44a1f22926fc923
3	uniswap_factory.vy	97d49145ec4fc6aa31099cb51c0c2f69b6e487b7
4	uniswap_exchange.vy	b3442cf7794d870511998ca3ba529d9ab42c4305

4.2 主要合约函数可见性分析

Contract Name	Function Name	Visibility
Distribute	Implementation	——
	setBeneficiary	Public
	setStartBlock	Public
	setStableHeight	Public
	setBlocksPerCycle	Public
	setReleasedPerBlock	Public
	revoke	Public

	release	Public
	releasableAmount	Public
AnyswapToken	Implementation	——
	destroy	Public
	burn	Public
	burnFrom	Public
	totalSupply	Public
	balanceOf	Public
	allowance	Public
	approve	Public
	transfer	Public
	transferFrom	Public
uniswap_factory	Implementation	——
	initializeFactory	Public
	createExchange	Public
	getExchange	Public
	getToken	Public
	getTokenWithId	Public
	Implementation	——
	setup	Public

uniswap_exchange	addLiquidity	Public
	removeLiquidity	Public
	getInputPrice	Private
	getOutputPrice	Private
	ethToTokenInput	Private
	default	Public
	ethToTokenSwapInput	Public
	ethToTokenTransferInput	Public
	ethToTokenOutput	Private
	ethToTokenSwapOutput	Public
	ethToTokenTransferOutput	Public
	tokenToEthInput	Private
	tokenToEthSwapInput	Public
	tokenToEthTransferInput	Public
	tokenToEthOutput	Private
	tokenToEthSwapOutput	Public
	tokenToEthTransferOutput	Public
	tokenToTokenInput	Private
	tokenToTokenSwapInput	Public
	tokenToTokenTransferInput	Public

	tokenToTokenOutput	Private
	tokenToTokenSwapOutput	Public
	tokenToTokenTransferOutout	Public
	tokenToExchangeSwapInput	Public
	tokenToExchangeTransferInput	Public
	tokenToExchangeSwapOutput	Public
	tokenToExchangeTransferOutput	Public
	transfer	Public
	transferFrom	Public
	approve	Public

4.3 代码审计详情

4.3.1 ERC777 合约重入风险

在进行代币兑换时未先扣除用户的代币，再进行转账操作。由于 ERC777 协议定义的标准中，在每次发生转账操作的时候都会去尝试调用代币发送者的 tokensToSend 函数，则交易合约在调用代币合约 transferFrom 函数时，代币合约将调用代币发送者的 tokensToSend 函数。此函数是代币发送者可控的函数，通过此函数可对交易合约的代币兑换函数进行二次调用造成重入风险。

代码位置： uniswap_exchange.vy 文件第 219、220、257、258、294、295、340 与 341 行

```
@private
def tokenToEthInput(tokens_sold: uint256, min_eth: uint256(wei), deadline: timestamp, buyer: address, recipient: address) ->
uint256(wei):
```

```
    assert deadline >= block.timestamp and (tokens_sold > 0 and min_eth > 0)
    tokens_fee: uint256 = (tokens_sold + 999) / 1000
    tokens_sold2: uint256 = tokens_sold - tokens_fee
    token_reserve: uint256 = self.token.balanceOf(self)
    eth_bought: uint256 = self.getInputPrice(tokens_sold2, token_reserve, as_unitless_number(self.balance))
    wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')
    assert wei_bought >= min_eth
    send(recipient, wei_bought)
    assert self.token.transferFrom(buyer, self.issuer, tokens_fee)
    assert self.token.transferFrom(buyer, self, tokens_sold2)
    log.EthPurchase(buyer, tokens_sold, wei_bought)
    return wei_bought
```

```
@private
def tokenToEthOutput(eth_bought: uint256(wei), max_tokens: uint256, deadline: timestamp, buyer: address, recipient:
address) -> uint256:
    assert deadline >= block.timestamp and eth_bought > 0
    token_reserve: uint256 = self.token.balanceOf(self)
    tokens_sold: uint256 = self.getOutputPrice(as_unitless_number(eth_bought), token_reserve,
as_unitless_number(self.balance))
    tokens_fee: uint256 = (tokens_sold + 999) / 1000
    tokens_sold2: uint256 = tokens_sold + tokens_fee
    # tokens sold is always > 0
    assert max_tokens >= tokens_sold2
    send(recipient, eth_bought)
    assert self.token.transferFrom(buyer, self.issuer, tokens_fee)
    assert self.token.transferFrom(buyer, self, tokens_sold)
    log.EthPurchase(buyer, tokens_sold2, eth_bought)
    return tokens_sold2
```

```
@private
def tokenToTokenInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline:
timestamp, buyer: address, recipient: address, exchange_addr: address) -> uint256:
    assert (deadline >= block.timestamp and tokens_sold > 0) and (min_tokens_bought > 0 and min_eth_bought > 0)
    assert exchange_addr != self and exchange_addr != ZERO_ADDRESS
    tokens_fee: uint256 = (tokens_sold + 999) / 1000
    tokens_sold2: uint256 = tokens_sold - tokens_fee
    token_reserve: uint256 = self.token.balanceOf(self)
    eth_bought: uint256 = self.getInputPrice(tokens_sold2, token_reserve, as_unitless_number(self.balance))
    wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')
    assert wei_bought >= min_eth_bought
    assert self.token.transferFrom(buyer, self.issuer, tokens_fee)
```

```
    assert self.token.transferFrom(buyer, self, tokens_sold2)
    tokens_bought: uint256 = Exchange(exchange_addr).ethToTokenTransferInput(min_tokens_bought, deadline, recipient,
value=wei_bought)
    log.EthPurchase(buyer, tokens_sold, wei_bought)
    return tokens_bought
```

```
@private
def tokenToTokenOutput(tokens_bought: uint256, max_tokens_sold: uint256, max_eth_sold: uint256(wei), deadline:
timestamp, buyer: address, recipient: address, exchange_addr: address) -> uint256:
    assert deadline >= block.timestamp and (tokens_bought > 0 and max_eth_sold > 0)
    assert exchange_addr != self and exchange_addr != ZERO_ADDRESS
    eth_bought: uint256(wei) = Exchange(exchange_addr).getEthToTokenOutputPrice(tokens_bought)
    eth_bought2:uint256(wei) = eth_bought * 1000 / 998 + 1
    token_reserve: uint256 = self.token.balanceOf(self)
    tokens_sold: uint256 = self.getOutputPrice(as_unitless_number(eth_bought2), token_reserve,
as_unitless_number(self.balance))
    tokens_fee: uint256 = (tokens_sold + 999) / 1000
    tokens_sold2: uint256 = tokens_sold + tokens_fee
    # tokens sold is always > 0
    assert max_tokens_sold >= tokens_sold2 and max_eth_sold >= eth_bought2
    assert self.token.transferFrom(buyer, self.issuer, tokens_fee)
    assert self.token.transferFrom(buyer, self, tokens_sold)
    eth_sold: uint256(wei) = Exchange(exchange_addr).ethToTokenTransferOutput(tokens_bought, deadline, recipient,
value=eth_bought2)
    log.EthPurchase(buyer, tokens_sold2, eth_bought2)
    return tokens_sold2
```

解决方案：在所有的代币兑换功能函数加入防重入机制，如 OpenZeppelin 的 ReentrancyGuard.sol，且在进行代币兑换的时，先扣除用户的代币，再将进行转账操作。

修复情况：经与项目方确认，所有接入代币都将进行上币审核，审核通过后由管理员更新至前端，后续将在上币审核时检查接入的代币合约是否有此风险。

4.3.2 Liquidity 无法移除风险

在 addLiquidityh 函数中使用了`assert self.token.transferFrom(msg.sender, self, token_amount)`，而在 removeLiquidity 函数中使用了`assert self.token.transfer(msg.sender, token_amount)`，两者不一致可能导致 Liquidity 无法移除的风险。例如：某合约的 transferFrom 的返回值符合 EIP 20 标准中定义的关于 ERC20 代币的返回值规范，而 transfer 的返回值不符合 EIP20 标准中定义的关于 ERC20 代币的返回值规范，则可能造成 addLiquidity 可以成功，但 removeLiquidity 无法成功的问题。

代码位置：uniswap_exchange.vy 文件第 62 行与第 96 行

```
@public
@payable
def addLiquidity(min_liquidity: uint256, max_tokens: uint256, deadline: timestamp) -> uint256:
    assert deadline > block.timestamp and (max_tokens > 0 and msg.value > 0)
    total_liquidity: uint256 = self.totalSupply
    if total_liquidity > 0:
        assert min_liquidity > 0
        eth_reserve: uint256(wei) = self.balance - msg.value
        token_reserve: uint256 = self.token.balanceOf(self)
        token_amount: uint256 = msg.value * token_reserve / eth_reserve + 1
        liquidity_minted: uint256 = msg.value * total_liquidity / eth_reserve
        assert max_tokens >= token_amount and liquidity_minted >= min_liquidity
        self.balances[msg.sender] += liquidity_minted
        self.totalSupply = total_liquidity + liquidity_minted
        assert self.token.transferFrom(msg.sender, self, token_amount)
        log.AddLiquidity(msg.sender, msg.value, token_amount)
        log.Transfer(ZERO_ADDRESS, msg.sender, liquidity_minted)
        return liquidity_minted
    else:
        assert (self.factory != ZERO_ADDRESS and self.token != ZERO_ADDRESS) and msg.value >= 1000000000
        assert self.factory.getExchange(self.token) == self
        token_amount: uint256 = max_tokens
        initial_liquidity: uint256 = as_unitless_number(self.balance)
        self.totalSupply = initial_liquidity
        self.balances[msg.sender] = initial_liquidity
        assert self.token.transferFrom(msg.sender, self, token_amount)
        log.AddLiquidity(msg.sender, msg.value, token_amount)
        log.Transfer(ZERO_ADDRESS, msg.sender, initial_liquidity)
```

```
        return initial_liquidity
```

```
@public
def removeLiquidity(amount: uint256, min_eth: uint256(wei), min_tokens: uint256, deadline: timestamp) -> (uint256(wei), uint256):
    assert (amount > 0 and deadline > block.timestamp) and (min_eth > 0 and min_tokens > 0)
    total_liquidity: uint256 = self.totalSupply
    assert total_liquidity > 0
    token_reserve: uint256 = self.token.balanceOf(self)
    eth_amount: uint256(wei) = amount * self.balance / total_liquidity
    token_amount: uint256 = amount * token_reserve / total_liquidity
    assert eth_amount >= min_eth and token_amount >= min_tokens
    self.balances[msg.sender] -= amount
    self.totalSupply = total_liquidity - amount
    send(msg.sender, eth_amount)
    assert self.token.transfer(msg.sender, token_amount)
    log.RemoveLiquidity(msg.sender, eth_amount, token_amount)
    log.Transfer(msg.sender, ZERO_ADDRESS, amount)
    return eth_amount, token_amount
```

解决方案：建议 addLiquidity 函数与 removeLiquidity 函数统一使用 transferFrom 函数进行转账操作。

修复情况： 经与项目方确认，所有接入代币都将进行上币审核，审核通过后由管理员更新至前端，后续将在上币审核时检查接入代币的 transferFrom 函数与 transfer 函数返回值是否符合 EIP 标准，以规避此风险。

4.3.3 使用 block.timestamp 获取当前时间

由于矿工可以对区块时间进行更改导致合约中使用 block.timestamp 获取到的时间可能并不准确。

代码位置：uniswap_exchange.vy 文件第 51、86、130、173、211、249、286 与 330 行

```
@public
@payable
def addLiquidity(min_liquidity: uint256, max_tokens: uint256, deadline: timestamp) -> uint256:
    assert deadline > block.timestamp and (max_tokens > 0 and msg.value > 0)
```

```
@public
```

```
def removeLiquidity(amount: uint256, min_eth: uint256(wei), min_tokens: uint256, deadline: timestamp) -> (uint256(wei), uint256):
    assert (amount > 0 and deadline > block.timestamp) and (min_eth > 0 and min_tokens > 0)
```

```
@private
def ethToTokenInput(eth_sold: uint256(wei), min_tokens: uint256, deadline: timestamp, buyer: address, recipient: address) -> uint256:
    assert deadline >= block.timestamp and (eth_sold > 0 and min_tokens > 0)
```

```
@private
def ethToTokenOutput(tokens_bought: uint256, max_eth: uint256(wei), deadline: timestamp, buyer: address, recipient: address) -> uint256(wei):
    assert deadline >= block.timestamp and (tokens_bought > 0 and max_eth > 0)
```

```
@private
def tokenToEthInput(tokens_sold: uint256, min_eth: uint256(wei), deadline: timestamp, buyer: address, recipient: address) -> uint256(wei):
    assert deadline >= block.timestamp and (tokens_sold > 0 and min_eth > 0)
```

```
@private
def tokenToEthOutput(eth_bought: uint256(wei), max_tokens: uint256, deadline: timestamp, buyer: address, recipient: address) -> uint256:
    assert deadline >= block.timestamp and eth_bought > 0
```

```
@private
def tokenToTokenInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline: timestamp, buyer: address, recipient: address, exchange_addr: address) -> uint256:
    assert (deadline >= block.timestamp and tokens_sold > 0) and (min_tokens_bought > 0 and min_eth_bought > 0)
```

```
@private
def tokenToTokenOutput(tokens_bought: uint256, max_tokens_sold: uint256, max_eth_sold: uint256(wei), deadline: timestamp, buyer: address, recipient: address, exchange_addr: address) -> uint256:
    assert deadline >= block.timestamp and (tokens_bought > 0 and max_eth_sold > 0)
```

解决方案：可以使用 oracles 进行时间获取。

修复情况：经与项目方确认，矿工能修改的区块时间是被限定在一个区块周期 15 秒以内。超过这个时间被认为是作恶，全网不会接受。交易合约中的时间限制精度是分钟级，默认是 15 分钟，1/60 的误差可以接受。

4.3.4 部分代码冗余

由于`Exchange(exchange).setup(token)`需要调用 exchange 合约的 setup 方法。因此，如果`exchangeTemplate`传入 0 地址，则将无法成功调用到 setup 方法，所以将无法成功创建交易合约。

代码位置：uniswap_factory.vy 文件第 15 行和第 21 行

```
@public
def initializeFactory(template: address):
    assert self.exchangeTemplate == ZERO_ADDRESS
    assert template != ZERO_ADDRESS
    self.exchangeTemplate = template
```

```
@public
def createExchange(token: address) -> address:
    assert token != ZERO_ADDRESS
    assert self.exchangeTemplate != ZERO_ADDRESS
    assert self.token_to_exchange[token] == ZERO_ADDRESS
    exchange: address = create_with_code_of(self.exchangeTemplate)
    Exchange(exchange).setup(token)
    self.token_to_exchange[token] = exchange
    self.exchange_to_token[exchange] = token
    token_id: uint256 = self.tokenCount + 1
    self.tokenCount = token_id
    self.id_to_token[token_id] = token
    log.NewExchange(token, exchange)
    return exchange
```

解决方案：删去不必要的代码。

修复情况：经与项目方确认，零地址检查逻辑对业务无影响，代码不做修改。

5. 审计结果

5.1 中危漏洞

- 重入风险

5.2 低危漏洞

- 移除流动性池设计缺陷
- 区块时间获取设计缺陷

5.3 增强建议

- 部分代码冗余

5.4 总结

审计结论：通过

审计编号：0X002007200001

审计时间：2020 年 07 月 20 日

审计团队：慢雾安全团队

审计总结：本次审计发现 4 个安全问题，经过沟通反馈确认审计过程中发现的风险均在可承受范围内。

6. 声明

慢雾仅就本报告出具前已经发生或存在的事实出具本报告，并就此承担相应责任。对于出具以后发生或存在的事实，慢雾无法判断其智能合约安全状况，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于信息提供者截至本报告出具时向慢雾提供的文件和资料(简称“已提供资料”)。慢雾假设：已提供资料不存在缺失、被篡改、删减或隐瞒的情形。如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，慢雾对由此而导致的损失和不利影响不承担任何责任。

慢雾科技
slow mist

官方网址

www.slowmist.com

电子邮箱

team@slowmist.com

微信公众号