Newton 安全审计报告

目录

1 前言(Executive Summary)

慢雾安全团队于 2019-07-18 日，收到牛顿团队对 Newton 安全审计申请，根据双方约定和项目特点制定审计方案，并最终出具安全审计报告。

慢雾安全团队采用“黑灰为主，白盒为辅”的策略，以最贴近真实攻击的方式，对项目方进行完整的安全测试。

慢雾科技区块链系统测试方法：

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	基于开源、未开源代码，对节点、SDK 等程序进行漏洞挖掘

慢雾科技区块链风险等级：

严重漏洞	严重漏洞会对区块链的安全造成重大影响，强烈建议修复严重漏洞。
高危漏洞	高危漏洞会影响区块链的正常运行，强烈建议修复高危漏洞。
中危漏洞	中危漏洞会影响区块链的运行，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区块链的操作，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 项目背景(Context)

2.1 项目简介

Newton 是在 Golang 版以太坊上构建的一条公链。

项目官网： https://www.newtonproject.org/

2.2 审计范围

本次安全审计的主要类型包括：

椭圆曲线签名算法

3 代码分析(Code Overview)

3.1 椭圆曲线签名算法

- crypto/signature_r1.go

```
// Copyright 2017 The go-ethereum Authors
// This file is part of the go-ethereum library.
//
// The go-ethereum library is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// The go-ethereum library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with the go-ethereum library. If not, see <http://www.gnu.org/licenses/>.
```

```
// +build !nacl,!js,!nocgo

package crypto

import (
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
    "errors"
    "fmt"
    "math/big"

    "github.com/ethereum/go-ethereum/log"
)

var (
    ErrInvalidMsgLen      = errors.New("invalid message length, need 32 bytes")
    ErrInvalidSignatureLen = errors.New("invalid signature length")
    ErrInvalidRecoveryID   = errors.New("invalid signature recovery id")
    ErrInvalidKey         = errors.New("invalid private key")
)

// Ecrecover returns the uncompressed public key that created the given signature.
func Ecrecover(hash, sig []byte) ([]byte, error) {
    //return secp256k1.RecoverPubkey(hash, sig)
    //ecRecovery2(messageHash []byte, sig []byte,recId int64) (*ecdsa.PublicKey, error)
    if len(hash) != 32 {
        return nil, ErrInvalidMsgLen
    }
    if err := checkSignature(sig); err != nil {
        return nil, err
    }
    recId := int64(sig[len(sig)-1])
    pubKey, err := ecRecovery2(hash, sig[:len(sig)-1], recId)
    if pubKey == nil {
        return nil, err
    }
    bk := elliptic.Marshal(S256(), pubKey.X, pubKey.Y)
    return bk, nil

}
```

```
// SigToPub returns the public key that created the given signature.
func SigToPub(hash, sig []byte) (*ecdsa.PublicKey, error) {
    if len(hash) != 32 {
        return nil, ErrInvalidMsgLen
    }
    if err := checkSignature(sig); err != nil {
        return nil, err
    }
    recId := int64(sig[len(sig)-1])
    pubKey, err := ecRecovery2(hash, sig[:len(sig)-1], recId)
    return pubKey, err
}

func checkSignature(sig []byte) error {
    if len(sig) != 65 {
        return ErrInvalidSignatureLen
    }
    if sig[64] >= 4 {
        return ErrInvalidRecoveryID
    }
    return nil
}

// DecompressPubkey parses a public key in the 33-byte compressed format.
func decompressPubkey2(x *big.Int, yBit byte) (*ecdsa.PublicKey, error) {
    if (yBit != 0x02) && (yBit != 0x03) {
        return nil, fmt.Errorf("invalid yBit")
    }
    if x == nil {
        return nil, fmt.Errorf("invalid x")
    }

    xx := new(big.Int).Mul(x, x)
    xxa := new(big.Int).Sub(xx, big.NewInt(3))
    yy := new(big.Int).Mul(xxa, x)
    yy.Add(yy, elliptic.P256().Params().B)
    yy.Mod(yy, elliptic.P256().Params().P)

    y1 := new(big.Int).ModSqrt(yy, elliptic.P256().Params().P)
    if y1 == nil {
        return nil, fmt.Errorf("can not revcovery public key")
```

```
    }

    getY2 := func(y1 *big.Int) *big.Int {
        y2 := new(big.Int).Neg(y1)
        y2.Mod(y2, elliptic.P256().Params().P)
        return y2
    }

    y := new(big.Int)

    if yBit == 0x02 {
        if y1.Bit(0) == 0 {
            y = y1
        } else {
            y = getY2(y1)
        }
    } else {
        if y1.Bit(0) == 1 {
            y = y1
        } else {
            y = getY2(y1)
        }
    }

    return &ecdsa.PublicKey{X: x, Y: y, Curve: elliptic.P256()}, nil
}
func ecRecovery2(messageHash []byte, sig []byte, recId int64) (*ecdsa.PublicKey, error)
{
    if recId < 0 || recId > 3 {
        return nil, fmt.Errorf("invalid value of v")
    }

    sigLen := len(sig)
    r := new(big.Int).SetBytes(sig[:(sigLen / 2)])
    s := new(big.Int).SetBytes(sig[(sigLen / 2):])
    if r.Cmp(secp256r1N) > 0 || s.Cmp(secp256r1N) > 0 {
        return nil, fmt.Errorf("r or s can not big then n")
    }

    p256 := elliptic.P256()
    n := p256.Params().N
    i := new(big.Int).SetInt64(recId / 2)
```

```
x := new(big.Int).Add(r, i.Mul(i, n))

prime := p256.Params().P
if x.Cmp(prime) > 0 {
    return nil, fmt.Errorf("x can not big then q")
}
yBit := byte(0x02)
if recId%2 == 0 {
    yBit = 0x02
} else {
    yBit = 0x03
}
R, err := decompressPubkey2(x, yBit)
if err != nil {
    return nil, err
}

r1, r2 := p256.ScalarMult(R.X, R.Y, n.Bytes())
zero := new(big.Int)
if !((r1.Cmp(zero) == 0) && (r2.Cmp(zero) == 0)) {
    return nil, fmt.Errorf("nR != point at infinity")
}

e := new(big.Int).SetBytes(messageHash)
eInv := new(big.Int).SetInt64(0)
eInv.Sub(eInv, e)
eInv.Mod(eInv, n)

rInv := new(big.Int).Set(r)
rInv.ModInverse(rInv, n)

srInv := new(big.Int).Set(rInv)
srInv.Mul(srInv, s)
srInv.Mod(srInv, n)

eInvrInv := new(big.Int).Mul(rInv, eInv)
eInvrInv.Mod(eInvrInv, n)

krx, kry := p256.ScalarMult(R.X, R.Y, srInv.Bytes())
kgx, kgy := p256.ScalarBaseMult(eInvrInv.Bytes())
kx, ky := p256.Add(krx, kry, kgx, kgy)
rkey := ecdsa.PublicKey{Curve: p256, X: kx, Y: ky}
```

```
    return &rkey, nil

}

// Sign calculates an ECDSA signature.
//
// This function is susceptible to chosen plaintext attacks that can leak
// information about the private key that is used for signing. Callers must
// be aware that the given hash cannot be chosen by an adversery. Common
// solution is to hash any input before calculating the signature.
//
// The produced signature is in the [R || S || V] format where V is 0 or 1.
func Sign(hash []byte, prv *ecdsa.PrivateKey) (sig []byte, err error) {
    if len(hash) != 32 {
        return nil, fmt.Errorf("hash is required to be exactly 32 bytes (%d)",
len(hash))
    }
    if prv == nil {
        return nil, ErrInvalidKey
    }
    //seckey := math.PaddedBigBytes(prv.D, prv.Params().BitSize/8)
    //defer zeroBytes(seckey)
    //return secp256k1.Sign(hash, seckey)

    // sign the hash
    r, s, err := ecdsa.Sign(rand.Reader, prv, hash)
    if err != nil {
        return nil, err
    }
    if s.Cmp(secp256r1halfN) > 0 {
        s = new(big.Int).Sub(secp256r1N, s)
    }

    // encode the signature {R, S}
    // big.Int.Bytes() will need padding in the case of leading zero bytes
    curveOrderByteSize := S256().Params().P.BitLen() / 8
    rBytes, sBytes := r.Bytes(), s.Bytes()
    signature := make([]byte, curveOrderByteSize*2+1)
    copy(signature[curveOrderByteSize-len(rBytes):], rBytes)
    copy(signature[curveOrderByteSize*2-len(sBytes):], sBytes)
    // TODO: fix v value.
    recId := byte(0)
```

```
    for recId = 0; recId < 4; recId++ {
        pk, _ := ecRecovery2(hash, signature[:len(signature)-1], int64(recId))
        if pk != nil && comparePublicKey(pk, &prv.PublicKey) == true {
            signature[len(signature)-1] = recId
            return signature, nil
        }
    }
    return nil, fmt.Errorf("could not construct a recoverable key. This should never
happen")
}

func comparePublicKey(key1, key2 *ecdsa.PublicKey) bool {
    x := key1.X.Cmp(key2.X)
    y := key2.Y.Cmp(key2.Y)
    if x == 0 && y == 0 {
        return true
    } else {
        return false
    }
}

// VerifySignature checks that the given public key created signature over hash.
// The public key should be in compressed (33 bytes) or uncompressed (65 bytes) format.
// The signature should have the 64 byte [R || S] format.
func VerifySignature(pubkey, hash, signature []byte) bool {
    //return secp256k1.VerifySignature(pubkey, hash, signature)
    if len(hash) != 32 {
        log.Info("hash length error")
        return false
    }
    if len(signature) != 64 {
        log.Info("signature length error")
        return false
    }
    if len(pubkey) == 0 {
        log.Info("public key length")
        return false
    }

    curveOrderByteSize := S256().Params().P.BitLen() / 8
    r, s := new(big.Int), new(big.Int)
```

```
    r.SetBytes(signature[:curveOrderByteSize])
    s.SetBytes(signature[curveOrderByteSize:])

    if len(pubkey) == 33 {
        publicKey, err := DecompressPubkey(pubkey)
        if err != nil {
            log.Info("decompress public key error")
            return false
        }
        return ecdsa.Verify(publicKey, hash, r, s)
    } else if (len(pubkey) == 65) && (pubkey[0] == 0x04) {
        x, y := elliptic.Unmarshal(S256(), pubkey)
        if x == nil || y == nil {
            log.Info("public key value error")
            return false
        }
        publicKey := ecdsa.PublicKey{Curve: S256(), X: x, Y: y}
        return ecdsa.Verify(&publicKey, hash, r, s)
    } else {
        log.Info("public key header error")
        return false
    }
}

// DecompressPubkey parses a public key in the 33-byte compressed format.
func DecompressPubkey(pubkey []byte) (*ecdsa.PublicKey, error) {
    if len(pubkey) != 33 {
        return nil, fmt.Errorf("invalid pubkey length")
    }
    yBit := pubkey[0]
    x := new(big.Int)
    x.SetBytes(pubkey[1:])
    return decompressPubkey2(x, yBit)
}

// CompressPubkey encodes a public key to the 33-byte compressed format.
func CompressPubkey(pubkey *ecdsa.PublicKey) []byte {
    //return secp256k1.CompressPubkey(pubkey.X, pubkey.Y)
    // big.Int.Bytes() will need padding in the case of leading zero bytes
    if pubkey == nil {
        return nil
    }
```

```
    curveOrderByteSize := S256().Params().P.BitLen() / 8
    xBytes := pubkey.X.Bytes()
    ckey := make([]byte, curveOrderByteSize+1)
    if pubkey.Y.Bit(0) == 1 {
        ckey[0] = 0x03
    } else {
        ckey[0] = 0x02
    }
    copy(ckey[1+curveOrderByteSize-len(xBytes):], xBytes)
    return ckey
}

// S256 returns an instance of the secp256k1 curve.
func S256() elliptic.Curve {
    return elliptic.P256()
}
```

3.2 分析小结

原版以太坊 Ecrecover、Sign 依赖于 C 语言版 secp256k1 算法和 btcec 算法两种，Newton 算法在此结构上进行修改，调用分析如下：

- *Ecrecover --> ecRecovery2 --> elliptic.Marshal 标准库将点编码*
- *SigToPub --> ecRecovery2 --> 得到公钥*
- *Sign --> ecdsa.Sign 标准库（安全性取决于 rand.Reader 随机程度） --> 返回 r s 填充到 sig --> 设置未位并检验可恢复性*
- *comparePublicKey --> 校验两个公钥是否相同*
- *VerifySignature --> 判断是何种编码格式的公钥，并解码 --> ecdsa.Verify（使用公钥验证 hash 值和两个大整数 r、s 构成的签名，并返回签名是否合法。）*
- *DecompressPubkey 将 33 字节格式公钥还原成公钥 --> decompressPubkey2*
- *CompressPubkey 将公钥编码成 33 字节格式*

从中我们可以看出：

- 签名与验证核心算法依赖 ecdsa 标准库
- ecRecovery2、decompressPubkey2 是改动比较多的算法

经沟通我们找到了改动算法 ecRecovery2 的参考文档，并进一步校对代码，未发现实现错误。

4 审计结果(Result)

4.1 模糊测试

对 decompressPubkey2 算法我们着重关注它的压缩和解压双向过程是否会出现数据失真，为此我们编写了测试用例。

```
func TestDecompress(t *testing.T) {
    for i := 1; i < 1000000; i++ {
        randkey, err := GenerateKey()
        if err != nil {
            t.Fatalf("randkey got %v", err)
        }
        dec := &randkey.PublicKey
        bk := CompressPubkey(dec)
        enc := hex.EncodeToString(bk)
        key, err := DecompressPubkey(bk)
        if err != nil {
            fmt.Println("failed in round:", i, enc)
            t.Fatalf("expected no error, got %v", err)
        }
        if comparePublicKey(key, dec) != true {
            t.Fatal("wrong result")
        }
    }
}
```

经大量测试，结果均为通过，未发现异常。

4.2 低危漏洞

- comparePublicKey 未对比 elliptic.Curve，攻击者可能通过构造的公钥绕过检查。

```
func comparePublicKey(key1, key2 *ecdsa.PublicKey) bool {
    x := key1.X.Cmp(key2.X)
    y := key2.Y.Cmp(key2.Y)
    if x == 0 && y == 0 {
        return true
```

```
    } else {
        return false
    }
}
```

4.3 增强建议

- 代码多处需要校验 hash 格式，建议编写单独函数用于 hash 校验。
- func TestUnmarshalPubkey 用例为原以太坊方法，需要更换。

4.4 结论

审计结果：通过

审计编号：BCA001907310002

审计日期：2019 年 07 月 31 日

审计团队：慢雾安全团队

综合结论：经反馈修正后，所有发现问题均已修复，综合评估 Newton 已无上述风险。

5 声明(Statement)

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，慢雾无法判断该项目安全状况，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。慢雾仅对该项目的安全情况进行约定内的安全审计并出具了本报告，慢雾不对该项目背景及其他情况进行负责。

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