ECDSA nonce reuse

장경배

https://www.youtube.com/watch?v=RJXD18x2QDU&feature=youtu.be

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

ECDSA

- **ECC**

  Elliptic Curve(타원곡선) 를 사용한 암호기술의 총체적인 이름이며 다음과 같은 용도로 사용됨.
  *양자내성암호는 아님

  유한체 상에서 정의한 타원곡선의 수학적 성질을 이용한 암호기술

  타원곡선이 암호학에 적합한 이유는, 실수상에서 연산과 유한체 상에서 연산에
  동일한 수학법칙이 적용되기 때문

  디지털 서명 용도 : ECDSA (Eliptic Curve Digital Signature Algorithm)
  키교환 용도 : ECDH (Elliptic Curve Diffie-Hellman)

- **ECDSA**

  타원곡선을 이용한 디지털 서명 알고리즘으로 현재 비트코인에서 사용되고 있음
  RSA 암호방식에 대한 대안으로 성능이 우수 → 3072-bit RSA 와 256-bit ECC의 암호성능이 동일

ECDSA

- ECDSA 에서 사용하는 방정식 (a, b 는 계수) → $y = x^3 + ax + b$
- ECC를 사용하기 위해선 타원곡선을 정의할 Parameter를 공유해야함 → 복잡한 Curve 상의 Point 연산이 필요 때문에 표준 단체에서는 타원곡선에 대한 몇가지 Parameter를 발표
- 비트코인에서 사용하는 secp256k1 curve 의 경우에는 a = 0 , b = 7 을 사용함

Parameter

Curve : 타원곡선 수식

g : 타원곡선의 기준점

n : g의 차수 , g 를 n 번 더했을 때 원점이 되는 값으로 충분히 큰 소수를 사용

ECDSA

Key Generation

Private key : d → 무작위로 선택된 1부터 n-1 사이의 정수

Public key : Q → Q = dg로 타원곡선의 더하기 연산으로 g를 d 번 더한 값
g는 공개된 값이지만 g를 안다고 해서 d값을 찾아내기는 매우 어려움

ECC에선 난수생성기가 매우 중요함

ECC는 Private key 의 bit 수가 적기 때문에 우수하지 않은
난수 생성기를 사용 할 경우 Private key 가 노출될 수 있음

→ 안드로이드 핸드폰의 비트코인 지갑이 해킹당한적이 있음
원인은 핸드폰에서 동작하는 난수 생성기를 사용하였기 때문

즉 Pirvate key 를 예측할 수 없는 Random한 난수 생성이 중요

ECDSA

Signature

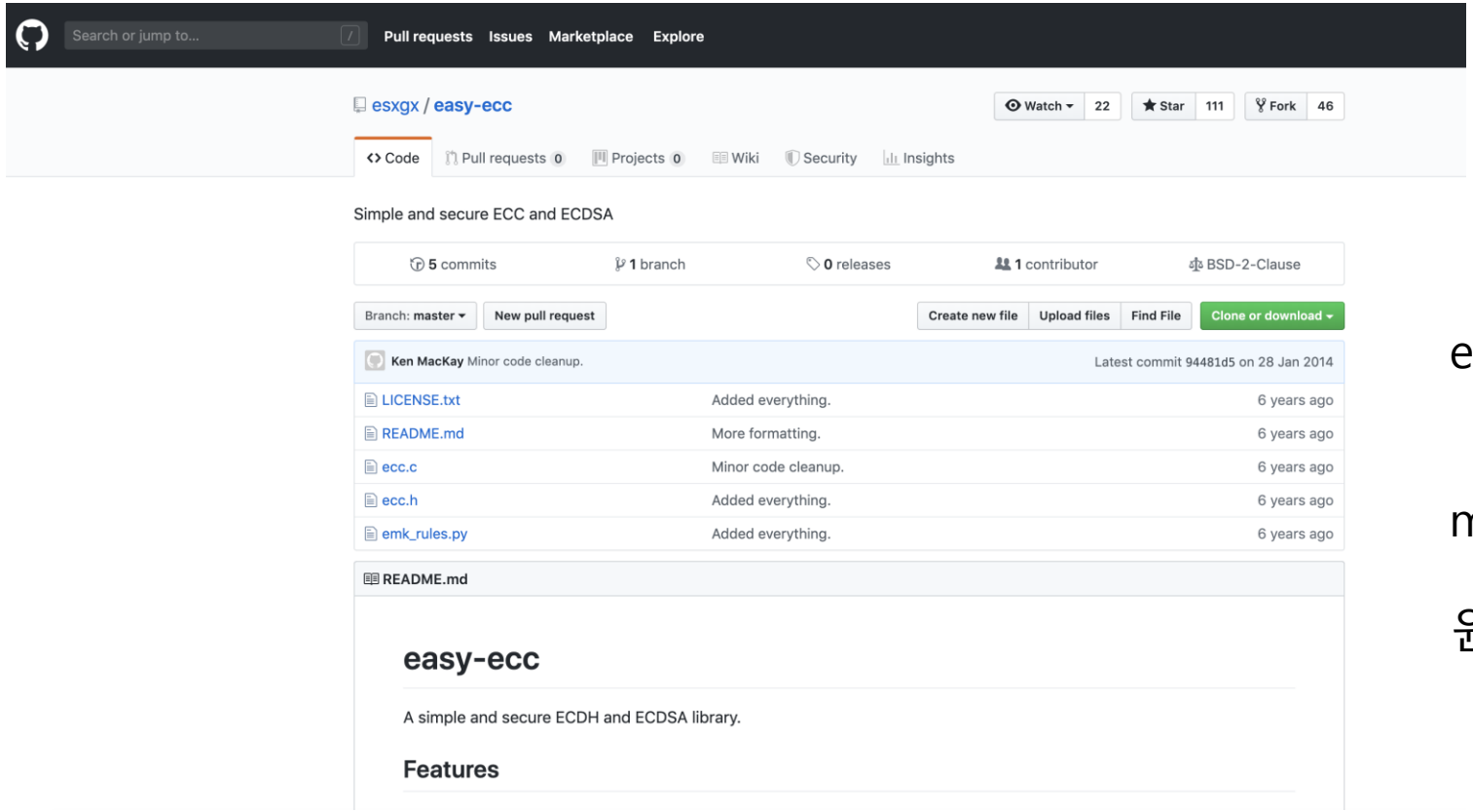
1. e = H(m)이고, H는 해시함수, m은 메세지
2. z는 e의 이진 값에서 왼쪽으로부터 n번째까지 잘라낸 값→ 길이 n
3. **암호학적으로 안전한 난수 k를 1 과 n-1 사이에서 생성 → nonce reuse vulnerability**
4. 곡선위의 점 (x, y) = k * g 를 계산
5. r = x (mod n)을 계산, 만약 r = 0 이면 3단계로 돌아가 다시 수행
6. s = $k^{-1}(z + rd)\ (mod\ n)$ 을 계산, 만약 s =0 이면 3단계로 돌아가 다시 수행 → 완성된 서명은 (r, s)

Signature Verify

1. r,s가 1부터 n-1사이의 정수인지 확인한다. 아니면 서명은 무효
2. e = H(m) , z는 e의 이진 값에서 왼쪽으로부터 n번째까지 잘라낸 값
3. $w = s^{-1}\ (mod\ n)$을 계산한 다음 $u1 = \ zw$, u2 = $\ rw\ (mod\ n)$
4. $(x, y) = u1\ *g + u2\ *Q$를 계산. 만약 $(x, y) = O$ 이면 서명은 무효
5. r $\equiv x1$ (mod n) 이면 유효한 서명

ECC Library

ECDSA library – https://github.com/esxgx/easy-ecc



ecc를 구현하기 위한 함수들이 제공됨

main 은 정의되어 있지 않아

원하는 함수 호출을 통해 구현 가능

ECC Library

```
#define CONCAT1(a, b) a##b
#define CONCAT(a, b) CONCAT1(a, b)

#define Curve_P_16 {0xFFFFFFFFFFFFFFFF, 0xFFFFFFFDFFFFFFFF}
#define Curve_P_24 {0xFFFFFFFFFFFFFFFFull, 0xFFFFFFFFFFFFFFFEull, 0xFFFFFFFFFFFFFFFFull}
#define Curve_P_32 {0xFFFFFFFFFFFFFFFFull, 0x00000000FFFFFFFFull, 0x0000000000000000ull, 0xFFFFFFFF00000001ull}
#define Curve_P_48 {0x00000000FFFFFFFF, 0xFFFFFFFF00000000, 0xFFFFFFFFFFFFFFFE, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF}

#define Curve_B_16 {0xD824993C2CEE5ED3, 0xE87579C11079F43D}
#define Curve_B_24 {0xFEB8DEECC146B9B1ull, 0x0FA7E9AB72243049ull, 0x64210519E59C80E7ull}
#define Curve_B_32 {0x3BCE3C3E27D2604Bull, 0x651D06B0CC53B0F6ull, 0xB3EBBD55769886BCull, 0x5AC635D8AA3A93E7ull}
#define Curve_B_48 {0x2A85C8EDD3EC2AEF, 0xC656398D8A2ED19D, 0x0314088F5013875A, 0x181D9C6EFE814112, 0x988E056BE3F82D19, 0xB3312FA7E23EE7E4}

#define Curve_G_16 { \
{0x0C28607CA52C5B86, 0x161FF7528B899B2D}, \
{0xC02DA292DDED7A83, 0xCF5AC8395BAFEB13}}

#define Curve_G_24 { \
{0xF4FF0AFD82FF1012ull, 0x7CBF20EB43A18800ull, 0x188DA80EB03090F6ull}, \
{0x73F977A11E794811ull, 0x631011ED6B24CDD5ull, 0x07192B95FFC8DA78ull}}

#define Curve_G_32 { \
{0xF4A13945D898C296ull, 0x77037D812DEB33A0ull, 0xF8BCE6E563A440F2ull, 0x6B17D1F2E12C4247ull}, \
{0xCBB6406837BF51F5ull, 0x2BCE33576B315ECEull, 0x8EE7EB4A7C0F9E16ull, 0x4FE342E2FE1A7F9Bull}}

#define Curve_G_48 { \
{0x3A545E3872760AB7, 0x5502F25DBF55296C, 0x59F741E082542A38, 0x6E1D3B628BA79B98, 0x8EB1C71EF320AD74, 0xAA87CA22BE8B0537}, \
{0x7A431D7C90EA0E5F, 0x0A60B1CE1D7E819D, 0xE9DA3113B5F0B8C0, 0xF8F41DBD289A147C, 0x5D9E98BF9292DC29, 0x3617DE4A96262C6F}}

#define Curve_N_16 {0x75A30D1B9038A115, 0xFFFFFFFE00000000}
#define Curve_N_24 {0x146BC9B1B4D22831ull, 0xFFFFFFFF99DEF836ull, 0xFFFFFFFFFFFFFFFFull}
#define Curve_N_32 {0xF3B9CAC2FC632551ull, 0xBCE6FAADA7179E84ull, 0xFFFFFFFFFFFFFFFFull, 0xFFFFFFFF00000000ull}
#define Curve_N_48 {0xECEC196ACCC52973, 0x581A0DB248B0A77A, 0xC7634D81F4372DDF, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF}
```

정의되어있는 타원곡선 파라미터 들

ECC Library

```c
int main(){
    uint8_t public[ECC_BYTES+1];
    uint8_t private[ECC_BYTES];
    uint8_t signature[ECC_BYTES*2];

    ecc_make_key(public,private);

    printf("\npublic :");
    for(int i=0; i<ECC_BYTES+1; i++)
        printf("%x ",public[i]);

    printf("\nprivate :");
    for(int i=0; i<ECC_BYTES; i++)
        printf("%x ",private[i]);
    return 0;
}
```

개인키와 공개키 쌍 생성

```
public :3 ae 1c 23 7c 12 b4 d2 8a 8 2b 9 4e 23 9a c2 f1 63 82 ab 70 3c ea
    8 fd 5c 68 c2 73 fa bb a3 68
private :16 38 e 11 90 c1 37 cf 5c 22 d6 b 16 75 d3 d7 8b 77 30 69 42 19
    1d 14 c6 61 9d cd 2e da f9 4f Program ended with exit code: 0
```

개인키 및 공개키 출력

ECDSA nonce reuse

Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies

Joachim Breitner[1][0000−0003−3753−6821] and Nadia Heninger[2]

[1] DFINITY Foundation, Zug, joachim@dfinity.org
[2] University of California, San Diego, nadiah@cs.ucsd.edu

Abstract. In this paper, we compute hundreds of Bitcoin private keys and dozens of Ethereum, Ripple, SSH, and HTTPS private keys by carrying out cryptanalytic attacks against digital signatures contained in public blockchains and Internet-wide scans. The ECDSA signature algorithm requires the generation of a per-message secret nonce. If this nonce is not generated uniformly at random, an attacker can potentially exploit this bias to compute the long-term signing key. We use a lattice-based algorithm for solving the hidden number problem to efficiently compute private ECDSA keys that were used with biased signature nonces due to multiple apparent implementation vulnerabilities.

Keywords: Hidden number problem, ECDSA, Lattices, Bitcoin, Crypto

레퍼런스

3.3 Elementary attacks on ECDSA

If an attacker learns the per-message nonce k used to generate an ECDSA signature, the long-term secret key d is easy to compute as $d = (sk - h)r^{-1} \bmod n$.

It is also well known that if the same nonce k is used to sign two different messages h_1 and h_2 with the same secret key, then the secret key is revealed. Let (r_1, s_1) be the signature generated on message hash h_1, and (r_2, s_2) be the signature on message hash h_2. We have immediately that $r_1 = r_2$, since $r_1 = r_2 = x(kG)$. Then we can compute $k = (h_1 - h_2)(s_1 - s_2)^{-1} \bmod n$, and recover the secret key as above.

개인키를 사용하여 같은 두개의 다른 메세지에 대한 서명을 생성할 때 같은 nonce k 를 사용하면 서명 시 사용 된 개인키가 노출된다는 내용

ECDSA nonce reuse

서명할 메세지는 "ECDSA" 와 "signature" 해시함수는 sha 256

```
int main(){
    uint8_t public[ECC_BYTES+1];
    uint8_t private[ECC_BYTES];
    uint8_t signature[ECC_BYTES*2];
    /* ECDSA 와 signature  메세지 해쉬 값(SHA-256)*/
    uint8_t hashECDSA[ECC_BYTES] = {0X38 ,0X9F ,0XA4 ,0X50 ,0X7C ,0XD5 ,0X36 ,0XC6 ,0X7D ,0XB3 ,0X5B ,0X80 ,0XB0 ,0X6A ,0XB0 ,0XB0 ,0XB0 ,0X34
        ,0XB7 ,0XA5 ,0XC6 ,0X7C ,0XF9 ,0XA2 ,0XD0 ,0X6E ,0XD0 ,0X08 ,0X76 ,0XD5 ,0X68 ,0XF9};
    uint8_t hashsignature[ECC_BYTES] = {0X1A ,0X2F ,0XC2 ,0X6D ,0XC7 ,0XEA ,0X5A ,0X2A ,0X47 ,0X48 ,0XB7 ,0XCB ,0X2B ,0X1E ,0XF1 ,0X93 ,0XD9
        ,0X6A ,0XB2 ,0XC9, 0X9F ,0X93 ,0X09 ,0X2F ,0X69 ,0XE6 ,0X30 ,0X75 ,0XB2 ,0X8D ,0X12 ,0X78};
    uint8_t newpublic[ECC_BYTES+1] = {0X03, 0XDC, 0X92, 0X13, 0XE6, 0XCD, 0XA0, 0X61, 0X95, 0X09, 0X8C, 0X90, 0X1C, 0X36, 0XC0, 0XF2, 0X0D,
        0X9B, 0X5D, 0XEC, 0XD2, 0X52, 0XF5, 0XF0, 0X0B, 0XFC, 0X5A, 0X63, 0XCA, 0XE0, 0XC5, 0XAE, 0XAC};

    /* 같은 개인키로 같은 nonce 값을 사용하여 만들어낸 "ECDSA"와 "signature" 의 서명*/
    uint64_t signature1[4] = {0X701660A50C6E3F17, 0XBFACE1FFE0868BD7 ,0X4FAE3BD85B9545BB, 0X09371E411284D26B};
    uint64_t signature2[4] = {0x19B3DB61F1680EDC, 0xC23CB3135D4A32CE, 0x5500186D83BFD1E1, 0xB84ABC62455C570D};
    /*ECDSA, signature 해쉬 값을 왼쪽부터 나열한 것*/
    uint64_t z1[4] = {0xd06ed00876d568f9, 0xb034b7a5c67cf9a2, 0x7db35b80b06ab0b0, 0x389fa4507cd536c6};
    uint64_t z2[4] = {0x69e63075b28d1278, 0xd96ab2c99f93092f, 0x4748b7cb2b1ef193, 0x1a2fc26dc7ea5a2a};
    //nonce값을 획득하고 서명하였을 때 나오는 r 값



    uint64_t tempsig[4];
    uint64_t tempz[4];
    uint64_t secretK[4]; //nonce k
    uint64_t sk[4];
    uint64_t privateKey[4]; // private key
    uint64_t storage[4];
```

ECDSA nonce reuse

3.3 Elementary attacks on ECDSA

If an attacker learns the per-message nonce k used to generate an ECDSA signature, the long-term secret key d is easy to compute as $d = (sk - h)r^{-1} \bmod n$.

It is also well known that if the same nonce k is used to sign two different messages h_1 and h_2 with the same secret key, then the secret key is revealed. Let (r_1, s_1) be the signature generated on message hash h_1, and (r_2, s_2) be the signature on message hash h_2. We have immediately that $r_1 = r_2$, since $r_1 = r_2 = x(kG)$. Then we can compute $k = (h_1 - h_2)(s_1 - s_2)^{-1} \bmod n$, and recover the secret key as above.

```
vli_modSub(tempz, z1, z2, curve_n); /* z2 - z1 */
vli_modSub(tempsig, signature1, signature2, curve_n); /*  s2 - s1 */
vli_modInv(tempsig, tempsig, curve_n); /*  1 / (s2 - s1) */
vli_modMult(secretK, tempsig, tempz, curve_n); /* k = (z2 - z1) / (s2- s1) */

printf("\n\n nonce k : %llx %llx %llx %llx \n",secretK[0],secretK[1],secretK[2],secretK[3]);
```

$$k = (h_1 - h_2)(s_1 - s_2)^{-1} \bmod n,$$

```
nonce k : 7777777777777777 7777777777777777
    7777777777777777 7777777777777777
```

nonce ***k***

ECDSA nonce reuse

```
int ecdsa_sign(const uint8_t p_privateKey[ECC_BYTES], const uint8_t p_hash[ECC_BYTES], uint8_t p_signature[ECC_BYTES*2])
{
    uint64_t k[NUM_ECC_DIGITS];
    uint64_t l_tmp[NUM_ECC_DIGITS];
    uint64_t l_s[NUM_ECC_DIGITS];
    EccPoint p;
    unsigned l_tries = 0;
    do
    {
        /* k에대한 난수 생성하는 과정
        if(!getRandomNumber(k) || (l_tries++ >= MAX_TRIES))
        {
            return 0;
        }

        if(vli_isZero(k))
        {
            continue;
        }

        if(vli_cmp(curve_n, k) != 1)
        {
            vli_sub(k, k, curve_n);
        }
         */

        k[0]=0x7777777777777777;
        k[1]=0x7777777777777777;
        k[2]=0x7777777777777777;
        k[3]=0x7777777777777777;

        /* tmp = k * G  곡선위의 점 (x1, y1) 계산  */
        EccPoint_mult(&p, &curve_G, k, NULL);

        /* r = x1 (mod n) 왼쪽 수식 말그대로 계산 r = 0이면 난수 다시 생성부터 다시 */
        if(vli_cmp(curve_n, p.x) != 1)
        {
            vli_sub(p.x, p.x, curve_n);
        }
    } while(vli_isZero(p.x));

    ecc_native2bytes(p_signature, p.x);

    ecc_bytes2native(l_tmp, p_privateKey);

    // s = z+rd / k 하는 과정

    vli_modMult(l_s, p.x, l_tmp, curve_n); /* s = r*d */

    ecc_bytes2native(l_tmp, p_hash);
    vli_modAdd(l_s, l_tmp, l_s, curve_n); /* s = e + r*d */

    vli_modInv(k, k, curve_n); /* k = 1 / k */
    vli_modMult(l_s, l_s, k, curve_n); /* s = (e + r*d) / k */


    ecc_native2bytes(p_signature + ECC_BYTES, l_s);

    return 1;
}
```

라이브러리의 ECDSA 서명 함수 → 기존 난수 생성 부분을 임의로 주석처리 후 난수 값 고정

ECDSA nonce reuse

3.3 Elementary attacks on ECDSA

If an attacker learns the per-message nonce k used to generate an ECDSA signature, the long-term secret key d is easy to compute as $d = (sk-h)r^{-1} \bmod n$.

It is also well known that if the same nonce k is used to sign two different messages h_1 and h_2 with the same secret key, then the secret key is revealed. Let (r_1, s_1) be the signature generated on message hash h_1, and (r_2, s_2) be the signature on message hash h_2. We have immediately that $r_1 = r_2$, since $r_1 = r_2 = x(kG)$. Then we can compute $k = (h_1 - h_2)(s_1 - s_2)^{-1} \bmod n$, and recover the secret key as above.

```
vli_modMult(sk, signature1, secretK, curve_n); /*  sk = s * k   */
vli_modSub(storage, sk, z1, curve_n); /* sk - z */
vli_modInv(R, R, curve_n);    /* r = 1 / r  */
vli_modMult(privateKey, storage, R, curve_n); /* privatekey = (sk - z) / r  */

printf("\n\nRetrived privateKey : %llx %llx %llx %llx \n",privateKey[0],privateKey[1],privateKey[2],privateKey[3]);
```

$$d = (sk-h)r^{-1} \bmod n.$$

```
Retrived privateKey : 8bd1c4fd1feabc15 97f7d26c97d4be07 6d704d1dd9e91362 e7c7589403578676
```

Private Key

ECDSA nonce reuse

```
Retrived privateKey : 8bd1c4fd1feabc15 97f7d26c97d4be07 6d704d1dd9e91362 e7c7589403578676
```

개인 키 획득

```
uint8_t newprivate[ECC_BYTES] = {0xe7,0xc7,0x58,0x94,0x03,0x57,0x86,0x76, 0x6d,0x70,0x4d,0x1d,0xd9,0xe9,0x13,0x62,
    0x97,0xf7,0xd2,0x6c,0x97,0xd4,0xbe,0x07, 0x8b,0xd1,0xc4,0xfd,0x1f,0xea,0xbc,0x15};
```

획득한 개인 키 생성

```
/*Sign "ECDSA" with retrived privatekey & Verify with Publickey */
ecdsa_sign(newprivate, hashECDSA, signature); /* ECDSA 서명 */
printf("\nsignature(ECDSA) : ");
for(int i=0; i<ECC_BYTES*2; i++)
printf("%x ", signature[i]);
printf("\n\nVerify Result : %d  [True(1) False(0)]\n",ecdsa_verify(newpublic, hashECDSA, signature)); /* 검증 */

/*Sign "signature" with retrived privatekey & Verify with Publickey */
ecdsa_sign(newprivate, hashsignature, signature); /* signature 서명 */
printf("\nSignature(signature) : ");
for(int i=0; i<ECC_BYTES*2; i++)
printf("%x ", signature[i]);
printf("\n\nVerify Result : %d  [True(1) False(0)]\n",ecdsa_verify(newpublic, hashsignature, signature)); /* 검증 */
```

획득한 개인키로 서명 후, 미리 공개되 있던 공개키로 서명을 검증

ECDSA nonce reuse

```
 nonce k : 7777777777777777 7777777777777777 7777777777777777 7777777777777777

Retrived privateKey : 8bd1c4fd1feabc15 97f7d26c97d4be07 6d704d1dd9e91362 e7c7589403578676

signature(ECDSA) : e4 50 54 eb 5b 1a bd 97 66 50 f7 f3 95 bf 51 d0 d8 dd 19 3e 1 74 e7 a1 4a 1c
    8c 12 7f bd f2 db 9 37 1e 41 12 84 d2 6b 4f ae 3b d8 5b 95 45 bb bf ac e1 ff e0 86 8b d7 70
    16 60 a5 c 6e 3f 17

Verify Result : 1  [True(1) False(0)]

Signature(signature) : e4 50 54 eb 5b 1a bd 97 66 50 f7 f3 95 bf 51 d0 d8 dd 19 3e 1 74 e7 a1 4a
    1c 8c 12 7f bd f2 db b8 4a bc 62 45 5c 57 d 55 0 18 6d 83 bf d1 e1 c2 3c b3 13 5d 4a 32 ce
    19 b3 db 61 f1 68 e dc

Verify Result : 1  [True(1) False(0)]
```

같은 nonce *k* 를 사용한 두개의 다른 메세지에 서명이 주어졌을 때 nonce 값 k 와 개인키 도출 후 검증

→ 공격자가 서명주인인 척 하고 메세지를 서명하면 다른사람들은 올바른 서명이라 판단