

Can't export EC kSecAttrTokenIDSecureEnclave public key

Hi all,

Using iOS 9 beta 2, I'm trying to export an elliptic curve public key that was generated with kSecAttrTokenIDSecureEnclave and kSecAccessControlPrivateKeyUsage but I am having a few issues.

26k

First, I can't specify kSecAttrIsPermanent for kSecPublicKeyAttrs or SecKeyGeneratePair() fails. I guess that makes sense because kSecAttrTokenIDSecureEnclave is specified for the entire SecKeyGeneratePair() operation (it fails if I put it under kSecPrivateKeyAttrs?) and there is no reason to save an elliptic curve public key with Secure Enclave protection. But this means that later looking up the elliptic curve public key with SecItemCopyMatching and kSecReturnData fail, so there doesn't seem to be a way to get the public key material in order to export the elliptic curve public key using the KeyChain API calls.

Second, of course I have the SecKeyRef for the elliptic curve public key returned by SecKeyGeneratePair(), but on iOS there is no way to export the elliptic curve public key from this opaque handle.

Third, SecKeyRef will print out diagnostic info for the elliptic curve public key though! This is the output for a typical elliptic curve public key as returned by the OS:

```
<SecKeyRef curve type: kSecECCurveSecp256r1, algorithm id: 3, key type: ECPublicKey, version: 3, block size: 256 bits, y: 0620A1AE78F7EA7D79F1CA6F63F5954BD710BDBCEA9F03838A5F939F60140A7E01, x: 120DE3D293CF8B6F8A6049942ABD2C206BC7050B2330C348FDBA2999A8CB1AD90620A1AE78F7EA7D79F1CA6F63F5954BD710BDBCEA9F03838A5F939F60140A7E01, addr: 0x134672110>
```


x and y are specified, so for the time being I thought I could export the elliptic curve public key from the x and y dump. But x is 130 hexadecimal digits and y is 66 hexadecimal digits? Shouldn't these values be 32 bytes each?

The Apple KeyChainTouchID sample from iOS 9 beta 2 does not show how to export elliptic curve public keys, only how to generate, sign, and delete.

Things work properly with RSA, but then kSecAttrTokenIDSecureEnclave and kSecAccessControlPrivateKeyUsage can't be specified.

Confused. Any help appreciated!

Security

Asked 5 years ago by [fijibill](#) 

[Reply to this question](#)

35 Replies

Hi jmarne,


Same here, I generated a keypair in openssl using the curve secp256r1, the one that's using SecKeyGeneratePair with kSecAttrKeyTypeEC. By trial and error I discovered that I had to add a standard ASN.1 OID header exactly like yours to the raw public key bytes returned by SecItemCopyMatching:

```
$ hexdump -C header_secp256r1.bytes
```

```
00000000 30 59 30 13 06 07 2a 86 48 ce 3d 02 01 06 08 2a |0Y0...*.H.=...*|
```

```
00000010 86 48 ce 3d 03 01 07 03 42 00                |.H.=....B.|
```

And then I'm able to use it with openssl. Now I'm trying to use this key to verify a signature sent by the iOS App, but I'm clueless on how to hash the data with the kind of signature hash algorithm that OpenSSL/PHP is waiting (that's "ecdsa-with-SHA1"). I have tried with openssl's generated keypair and it's working, so apparently there should be a way to doing the same with the result of SecKeyRawSign.

Posted 5 years ago by [Digital Leaves](#) 

Digital Leaves, Sorry but I don't know anything about the openssl/php library. However, after a quick google search, I'm guessing that you need to pass in this hash algorithm: "[14] => ecdsa-with-SHA1" as documented on this page: <http://php.net/manual/en/function.openssl-get-md-methods.php> to the verify function. Also, I'm not sure you have to re-hash the original data during verification in PHP. If I'm reading this function correctly, <http://php.net/manual/en/function.openssl-verify.php>. You just have to pass the original data that was used to generate the signature along with the hash algorithm used and php will hash it for you during the verify call.

As an FYI, my ANS1 Dump of the iOS ECC public key produces this OID:

```
ObjectIdentifier(1.2.840.113549.1.1.1)
```


Which appears to correspond to an RSA encryption header (according to RFC 5480 <https://tools.ietf.org/html/rfc5480>).

```
-- RSA PK Algorithm and Key
```

```
rsaEncryption OBJECT IDENTIFIER ::= {
```

```
    iso(1) member-body(2) us(840) rsads(113549) pkcs(1) pkcs-1(1) 1 }
```

So, if I'm guessing correctly, we're getting an RSA header on an ECC public key which is reason for the need to swap out OID headers.


Posted 5 years ago by [jmarne](#) 

Hi Jmarne,

Thank you. In my opinion, for some reason, Apple does not store the public key header information. I guess this is for the same reason that they only allow ECC keys when using secure enclave, the size of the keys (ECC keys are way shorter and space-efficient than, say, RSA ones). The header actually corresponds to a header for a secp256r1 curve, only they won't include that, which is very unfortunate, and leave us guessing how to transform/use that key.

The problem with the signing process (for me at least) is that the signature usually occurs in conjunction with a hash of the data, not directly to the data, but I have no way of knowing if SecKeyRawSign is doing that for ECC keys, and if so, if it's using what openssl will call a SHA1 hash algorithm or the ecdsa-with-SHA1 one (I can't find any ECC reference in all the CC_SHAXXX functions in CC). I will definitely have to check out, but due to signature verification being a blackbox (0-1), and the lack of Apple documentation regarding EC key signing-hashing process, I will need to spend some time to make it work I guess.

Anyway thanks a lot, that info really helps me to verify that I'm in the right track.

Posted 5 years ago by [Digital Leaves](#) 

At the end, I finally managed to solve this. Meaning, I am able to generate the key pair, protecting the private key in the secure enclave and touch id, export the public key, sign data using the private key and finally verify the signature using openssl.

As described in the posts above, the public key is missing the header bytes, so those need to be added.

Now when signing the data on the device, I did this:

```
SecKeyRawSign(privateKey, kSecPaddingSigRaw, hashBytes, hashBytesSize, signedHashBytes,
&signedHashBytesSize);
```


hashBytes: this is the SHA1 hash of the data I want to sign.

Now to verify I use openssl like this:

```
openssl dgst -ecdsa-with-SHA1 -verify pubkey.pem -signature signature.bin data.txt
```

signature.bin: this just contains the output of SecKeyRawSign(), no manipulation needed.

Hope this is helpful.

Posted 5 years ago by [csgodenzim](#) 

Hi,

Thanks csgodenzim, I tried with that padding also (all paddings, actually without success). Is there a way I can talk to you by email or something?


Are you doing a Hash by using CC_SHA1 of the data in iOS prior to calling SecKeyRawVerify? I'm getting mad trying to make this work. Here is my code. I tried "ecdsa-with-SHA1", "SHA1", etc at the server side with no luck (well, I think I tried everything).

```
if let privateKeyRef = self.getPrivateKeyReference() { // <==== Private key is valid, EC key ref
from secure enclave (checked)
    let resultData = NSMutableData(length: LargeEnoughBufferSize)!
    let resultPointer = UnsafeMutablePointer<UInt8>(resultData.mutableBytes)
    var resultLength = resultData.length

    if let plainData = message.dataUsingEncoding(NSUTF8StringEncoding) {
        let hashData = NSMutableData(length: Int(CC_SHA1_DIGEST_LENGTH))!
        let hash = UnsafeMutablePointer<UInt8>(hashData.mutableBytes)
        CC_SHA1(UnsafePointer<Void>(plainData.bytes), CC_LONG(CC_SHA1_DIGEST_LENGTH), hash)

        let status = SecKeyRawSign(privateKeyRef, SecPadding.PKCS1, hash, hashData.length,
resultPointer, &resultLength) // <== SigRaw shows equal results
        if status != errSecSuccess {
            ... // handle error.
        } else { // <===== Status is errSecSuccess, this is reached and resultData is shown
and looks valid.
            resultData.length = resultLength
            print("Result: \(status). Generated result: \(resultData)")
        }
    } else { ... } // handle error
    // <===== now send resultData to server. Server is unable to get it.
} else { ... } // handle error
```

Please, any help REALLY appreciated. Thanks in advance.

Posted 5 years ago by [Digital Leaves](#) 

A developer wrote into DTS asking about Secure Enclave cryptography, which gave me the opportunity to spend some time looking at this. Let's explain what's going on here.

When you generate a key pair as per the [KeychainTouchID](#) sample code, the private key gets store in the Secure Enclave leaving the public key represented solely in memory. You can add that public key to the keychain with code like this:

```
let addErr = SecItemAdd([
    kSecClass as String: kSecClassKey,
    kSecValueRef as String: publicKey,
    kSecAttrApplicationTag as String: uuidStr
] as NSDictionary, nil)
```

Note In my case I wanted to tag the key with a UUID so that I could come back to it later.

If you then do a

```
SecItemCopyMatching
```

on the public key to get back some data (via

```
kSecReturnData
```

), you see something like this:

```
04
B24DB122 E2DDDC97 FB0F58ED 7836F2CA
6868C040 6B483FDA 473FDD00 41D380B0
4EDB6183 3273046A 33D7A9A8 D66B93AA
A10B8732 93C68114 9CC7FA5E CD3523CE
```

This is

```
secp256r1
```

public key as defined by:

- SEC 1 [Elliptic Curve Cryptography](#)
- SEC 2 [Recommended Elliptic Curve Domain Parameters](#)

The latter standard shows examples that look just like this (see Section 2.4.2).

Notably, the leading 04 indicates that the key is uncompressed. You could potentially see a 02 or 03 in the same place, indicating a compressed key. Either way, that first byte is a good way to detect that you're seeing the data you're expecting to see.

When you put such a key into a certificate, you have to wrap it in a

```
SubjectPublicKeyInfo
```

ASN.1 structure. That wrapping is defined by RFC 5480 "[Elliptic Curve Cryptography Subject Public Key Information](#)". The structure looks like this:

```
SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm      AlgorithmIdentifier,
    subjectPublicKey BIT STRING
}
```

where

```
subjectPublicKey
```

is the

```
secp256r1
```

public key as discussed earlier and

```
algorithm
```

is defined like this:

```
AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY DEFINED BY algorithm OPTIONAL
}
```

Its

```
algorithm
```

OID is always 1.2.840.10045.2.1 (that is,

```
id-ecPublicKey
```

) and there's a single key parameter, another OID, 1.2.840.10045.3.1.7, or

```
secp256r1
```

, which identifies a

```
secp256r1
```

public key.

IMPORTANT iOS tends to deal with raw keys whereas most other security toolkits, like OpenSSL, tend to deal with public keys wrapped in a

```
SubjectPublicKeyInfo
```

structure.

Now, for a given EC key type the first N bytes of the

```
SubjectPublicKeyInfo
```

are fixed, hence the

```
headerBytes
```

array that jmarne uncovered earlier. If you take those header bytes:

```
30593013 06072A86 48CE3D02 0106082A
8648CE3D 03010703 4200
```

and slap them on the front of a

```
secp256r1
```

public key.

```
04
B24DB122 E2DDDC97 FB0F58ED 7836F2CA
6868C040 6B483FDA 473FDD00 41D380B0
4EDB6183 3273046A 33D7A9A8 D66B93AA
A10B8732 93C68114 9CC7FA5E CD3523CE
```

you end up with a file that you can dump with [dumpasn1](#):

```
$ dumpasn1 -p tmp.asn1
SEQUENCE {
  SEQUENCE {
    OBJECT IDENTIFIER '1 2 840 10045 2 1'
    OBJECT IDENTIFIER ansiX9p256r1 (1 2 840 10045 3 1 7)
  }
  BIT STRING
    04 B2 4D B1 22 E2 DD DC 97 FB 0F 58 ED 78 36 F2
    CA 68 68 C0 40 6B 48 3F DA 47 3F DD D0 41 D3 80
    B0 4E DB 61 83 32 73 04 6A 33 D7 A9 A8 D6 6B 93
    AA A1 0B 87 32 93 C6 81 14 9C C7 FA 5E CD 35 23
    CE
}
```

And, lo!, a nicely wrapped EC public key suitable for use with OpenSSL and so on.

WARNING This 'prefix with a fixed header' approach only works if the public key is of the right type and is *not* compressed. In an ideal world you would build up the

```
SubjectPublicKeyInfo
```

structure properly, which would allow you to work with both compressed and uncompressed keys. That would, however, require you to write some ASN.1 generation code.

Share and Enjoy — Quinn "The Eskimo!" Apple Developer Relations, Developer Technical Support, Core OS/Hardware

```
let myEmail = "eskimo" + "1" + "@apple.com"
```

Posted 5 years ago by [eskimo](#)  

hi Eskimo,

Thank you so very much for this comprehensive explanation. it really makes a lot more sense now. I suspect the signature generated by the SecKeyRawSign has a similar issue with the wrapping that prevents it for being properly verified from OpenSSL, which is expecting some kind of ASN.1 wrapping?

thanks again.

Posted 5 years ago by [Digital Leaves](#) 

I suspect the signature generated by the SecKeyRawSign has a similar issue with the wrapping that prevents it for being properly verified from OpenSSL, which is expecting some kind of ASN.1 wrapping?

When last I looked at *that* side of things I definitely had fun mapping OpenSSL's padding options to those used by our Security framework. I eventually got it working, see the [CryptoCompatibility](#) sample code, but that was for RSA, not EC.

Share and Enjoy — Quinn "The Eskimo!" Apple Developer Relations, Developer Technical Support, Core OS/Hardware

```
let myEmail = "eskimo" + "1" + "@apple.com"
```

Posted 5 years ago by [eskimo](#)  

When last I looked at *that* side of things I definitely had fun mapping OpenSSL's padding options to those used by our Security framework. I eventually got it working, see the [CryptoCompatibility](#) sample code, but that was for RSA, not EC.

I had cause to play around with this today and managed to implement an end-to-end test from iOS to OpenSSL:

1. iOS app generates an EC key pair with the private key in the Secure Enclave
2. iOS app exports the public key
3. iOS app signs some data with the private key and prints the data, the signature, and the public key
4. on the Mac I put that data into various files
5. I prepended the SubjectPublicKeyInfo to the raw EC public key
6. I converted the public key to PEM (see below)
7. I then verified the public key with OpenSSL's command line tool (see below)

```
$ openssl ec -pubin -inform DER -in EC\ key.asn1 -pubout -outform PEM -out EC\ key.pem
read EC key
writing EC key
```

```
$ openssl dgst -ecdsa-with-SHA1 -verify EC\ key.pem -signature signature.dat dataToSign.dat
Verified OK
```

While working on this I encountered two gotchas:

- When verifying the signature, you have to use

```
-ecdsa-with-SHA1
```

because OpenSSL treats

```
-sha1
```

as implying RSA.

- When calling

```
SecKeyRawSign
```

with an EC key, you have to use a padding of

```
kSecPaddingPKCS1
```

because

```
kSecPaddingPKCS1SHA1
```

implies RSA (probably for the same reason).

Share and Enjoy — Quinn "The Eskimo!" Apple Developer Relations, Developer Technical Support, Core OS/Hardware

```
let myEmail = "eskimo" + "1" + "@apple.com"
```

Posted 5 years ago by [eskimo](#)  

Hi Eskimo,

Thanks. Unfortunately, it's not working on my side. I knew that I had to use ecdsa-with-SHA1, and tried with both PKCS1SHA1 and PKCS1 with no luck.

I have the following scenario:

1. I generate an EC keypair with secure enclave, private key is held there, public key is extracted as NSData after adding to the keychain, pre-pended with the SubjectPublicKeyInfo containing the ASN.1 data, and sent to the server.

```
let privateKeyParams: [String: AnyObject] = [
    kSecAttrAccessControl as String: accessControl,
    kSecAttrIsPermanent as String: true,
    kSecAttrApplicationTag as String: "privateAppTag"
]

let publicKeyParams: [String: AnyObject] = [
    kSecAttrApplicationTag as String: "publicAppTag"
]

let parameters: [String: AnyObject] = [
    kSecAttrKeyType as String: kSecAttrKeyTypeEC,
    kSecAttrKeySizeInBits as String: 256,
    kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave,
    kSecPublicKeyAttrs as String: publicKeyParams,
    kSecPrivateKeyAttrs as String: privateKeyParams
]

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)) { () -> Void in
    var pubKey, privKey: SecKeyRef?
    let status = SecKeyGeneratePair(parameters, &pubKey, &privKey)
    if status == errSecSuccess {
        // add public key to keychain, prepend SubjectPublicKeyInfo to raw data and send to the
server.
    } else {
        // handle error
    }
}
```

2. The server receives the key, properly parses it and converts it to PEM. I know the key is successfully processed because openssl recognizes it:

```
$ openssl ec -pubin -noout -text -in public_key.pem
read EC key
pub:
    04:1f:48:a0:df:0f:ca:c9:95:59:91:65:e2:d0:bf:
    4c:03:53:a0:f7:c1:a4:e6:c4:37:e9:bc:0a:af:1e:
    ab:a2:9a:1d:5f:33:1a:fb:e5:30:e8:7e:b6:49:e1:
    01:2c:83:7a:55:39:87:4c:97:3d:f3:a0:c3:51:
    b6:ad:12:77:58
ASN1 OID: prime256v1
```

This is the ASN.1 dump of the key:

```
$ openssl asn1parse -in public_key.pem
    0:d=0  hl=2  l= 89 cons: SEQUENCE
      2:d=1  hl=2  l= 19 cons: SEQUENCE
        4:d=2  hl=2  l=  7 prim: OBJECT                :id-ecPublicKey
        13:d=2  hl=2  l=  8 prim: OBJECT                :prime256v1
        23:d=1  hl=2  l= 66 prim: BIT STRING
```

3. I sign a plain text (from the iOS side) to send to the server for verification. I use SecKeyRawSign with SecPadding.PKCS1. Data is a SHA1 of the plain text using the CC_SHA1 function.

```
signing plain text: haGXtJfQBLPmUICW+ArdtOiBW4Yckv8hCJhhrGKcwo8yj46o0YgaQg==
Result: 0. Generated signature: <30460221 00ed3648 6788fa05 1eff6b3b 0f09438a 4032a358 2bcd37b9 f3db9429
497a12f6 7e022100 a4ff57e1 6cb763c7 a7f2f0d4 b44add73 e939019c 4eb75df6 36590f22 2ce29258>
```

After that I do a verification in iOS that passes OK (just to be sure). I use the same CC_SHA1 function to get the digest and the same padding.

```
Verifying signature...
Text to verify: haGXtJfQBLPmUICW+ArdtOiBW4Yckv8hCJhhrGKcwo8yj46o0YgaQg==
Signature to verify: Optional(<30460221 00ed3648 6788fa05 1eff6b3b 0f09438a 4032a358 2bcd37b9 f3db9429
497a12f6 7e022100 a4ff57e1 6cb763c7 a7f2f0d4 b44add73 e939019c 4eb75df6 36590f22 2ce29258>)
Verifying self signature with public key:
text as data: <68614758 744a6651 424c506d 556c4357 2b417264 744f6942 57345963 6b763868 434a6868 72474b63
776f3879 6a34366f 30596761 51673d3d>
signature as data: <30460221 00ed3648 6788fa05 1eff6b3b 0f09438a 4032a358 2bcd37b9 f3db9429 497a12f6 7e022100
a4ff57e1 6cb763c7 a7f2f0d4 b44add73 e939019c 4eb75df6 36590f22 2ce29258>
Key: <SecKeyRef curve type: kSecECCurveSecp256r1, algorithm id: 3, key type: ECPublicKey, version: 3, block
size: 256 bits, y: 587712ADB651C3A0F33D974C873939557A832C01E149B67EE830E5FB1A335F1D01, x:
9AA2AB1EAF0ABCE937C4E6A4C1F7A053034CBFD0E265915995C9CA0FDA0481F587712ADB651C3A0F33D974C873939557A832C01E149B
67EE830E5FB1A335F1D01, addr: 0x135a20870>
Status: 0 (errSecSuccess)
```

4. However, if I get the signature data, copy and paste it on a binary file, and verify it with openssl (same command as you used):

```
openssl dgst -ecdsa-with-SHA1 -verify public_key.pem -signature signature.bin plaintext.txt

Verification Failure
```

If I generate the keypair using openssl, sign and verify, everything goes fine:

```
1. create the private EC key w/ public key.
openssl ecparam -genkey -name prime256v1 -noout -out myprivatekey.pem
2. generate the public EC key from the private EC key
openssl ec -in myprivatekey.pem -pubout -out mypubkey.pem
3. Sign the plaintext.txt file
openssl dgst -ecdsa-with-SHA1 -sign myprivatekey.pem plaintext.txt > signature.bin
4. Verify signature
openssl dgst -ecdsa-with-SHA1 -verify mypubkey.pem -signature signature.bin plaintext.txt

Verified OK
```

The Swift signing code is really simple, I think, and I don't know where the problem may be:

```
let resultData = NSMutableData(length: 1024)!
let resultPointer = UnsafeMutablePointer<UInt8>(resultData.mutableBytes)
var resultLength = resultData.length

if let plainData = message.dataUsingEncoding(NSUTF8StringEncoding) {
    /
    let hashData = NSMutableData(length: Int(CC_SHA1_DIGEST_LENGTH))!
    let hash = UnsafeMutablePointer<UInt8>(hashData.mutableBytes)
    CC_SHA1(UnsafePointer<Void>(plainData.bytes), CC_LONG(CC_SHA1_DIGEST_LENGTH), hash)

    let status = SecKeyRawSign(privateKeyRef, SecPadding.PKCS1, hash, hashData.length, resultPointer,
    &resultLength)
    if status != errSecSuccess {
        error = .UnableToEncrypt
    } else {
        resultData.length = resultLength
        print("Result: \(status). Generated signature: \(resultData)")
        // base64 encode resultData and send to the server for verification.
    }
} else { error = .WrongInputDataFormat }
```

I really don't know what can be wrong or how to get out of this. Thanks in advance for your time.

Best


Posted 5 years ago by [Digital Leaves](#) 

Unfortunately, it's not working on my side.

Bummer. I couldn't see anything obviously wrong with your technique. Please drop me a line at my individual email address (in my signature, below).

Share and Enjoy — Quinn "The Eskimo!" Apple Developer Relations, Developer Technical Support, Core OS/Hardware

```
let myEmail = "eskimo" + "1" + "@apple.com"
```

Posted 5 years ago by [eskimo](#)  

Hi Eskimo.

I am having the same problem her. I have done

- Created the key in the secure element.
- Made a signature with kSecPaddingPKCS1 of a SHA1 hashed plain text.
- Verified it on the Iphone sucessfully.
- Exported the public key sucessfully with the header.(Looks OK using a ASN.1 dump)

And I am completely unable to verify this signature using either openssl nor a java program....

Would you mind sharing the data you are doing your experiment on ? You are not ecplctly stading that you do make a SHA-1 hash before signing and that you place the plaintext into the file for openssl validation, but I assume you are ?

I.e. what do you sign ?

And do you make any special provisions when putting this on a file for openssl verification (Like to compensate for end of file marker ?)

In addition what version of open SSL are you using ?

Thaks for your time...

Ronny

Posted 5 years ago by [Ronnyk](#) [🔗](#)

You are not ecplctly stading that you do make a SHA-1 hash before signing and that you place the plaintext into the file for openssl validation, but I assume you are ?

Correct. In the example I posted:

- `EC key.pem`
is the concatenation of the 'standard' EC header,
`EC key header.dat`
, and the EC key from iOS,
`EC key raw.dat`
- `signature.dat`
was the output from
`SecKeyRawSign`
on iOS
- `dataToSign.dat`
was a bunch of random bytes that's my test 'message'

The paste dumps are pasted in below.

And do you make any special provisions when putting this on a file for openssl verification (Like to compensate for end of file marker ?)

No.

In addition what version of open SSL are you using ?

The one built in to OS X 10.11.

Share and Enjoy — Quinn "The Eskimo!" Apple Developer Relations, Developer Technical Support, Core OS/Hardware

```
let myEmail = "eskimo" + "1" + "@apple.com"
```

```
$ hexdump -v EC\ key\ header.dat
00000000 30 59 30 13 06 07 2a 86 48 ce 3d 02 01 06 08 2a
00000010 86 48 ce 3d 03 01 07 03 42 00
```

```
$ hexdump -v EC\ key\ raw.dat
00000000 04 5a e5 3a da 6e b3 21 53 a4 0e 6b de bc 54 bb
00000010 6d 89 c2 6c 22 8a 6a 82 47 96 57 5a e3 a0 a4 dc
```

```
0000020 17 0c fb ce 4a f2 ed 34 a5 62 77 04 f7 b9 2e 5b
0000030 38 61 74 81 c0 a3 62 63 79 74 0e 67 af dd df 74
0000040 ab
```

```
$ hexdump -v signature.dat
0000000 30 44 02 20 09 57 a8 25 a4 60 79 3b 00 24 06 81
0000010 81 7a c6 f8 81 29 96 65 ca dc da 7d f6 15 ef c2
0000020 68 d2 b9 d9 02 20 22 05 cd d8 a8 28 91 85 9c 31
0000030 1f c8 a4 57 cc 94 42 68 73 94 05 fd 54 b3 ba 39
0000040 16 a4 eb c0 8a 9c
```

```
$ hexdump -v dataToSign.dat
0000000 2f 8d e6 68 42 19 d4 5b e2 e3 30 c8 c6 2c e1 0b
0000010 69 4c 9c af c4 82 ee d5 67 e0 15 49 72 c8 60 4a
0000020 4b 31 2d 9f 94 89 c5 49 f3 6e b9 ab b7 ae 76 52
0000030 5a 56 d8 c4 c6 15 17 50 db 72 83 f4 c3 76 f8 fe
0000040 8e db 8a b5 6c 62 7b 5f 33 87 57 47 53 1c 69 b5
0000050 f5 e9 40 fc c8 fb d0 62 76 51 13 c5 8e fd b9 42
0000060 dc a9 c7 68 47 ea dc 4f 7f b1 f5 04 fb 53 a3 96
0000070 f4 58 88 9d 2d fd 13 ff 0a b4 d7 2b 5f f4 f2 ce
0000080 9f 31 25 d9 fa 86 27 2e 71 36 ce 99 8a c3 0e b8
0000090 9b 1c d6 be 63 3c a7 6d db 6e 7c da 53 40 83 03
00000a0 ec f5 d3 fb e9 1c ad a9 b1 5f 9b dd 79 d7 45 d2
00000b0 16 0c c9 2f 62 a6 78 99 8d ea fb 50 21 42 90 81
00000c0 2b 7f 3e d7 56 52 43 9c 81 21 40 39 50 7d d6 a6
00000d0 b9 d9 1c a7 b6 0c 05 95 5c f5 1e 20 84 ce f9 5e
00000e0 89 00 83 e9 08 7f 3c 96 f0 f6 90 e3 d4 e8 7c 4e
00000f0 02 28 8a bf ba 2d ba 91 dc 67 f8 6d bd 50 ef fe
0000100 db 31 5e 05 6f cc b8 26 b7 2c 3e aa f2 32 56 5b
0000110 18 ae 98 d0 a5 27 85 62 78 49 51 79 88 05 5d e8
0000120 10 dc 79 fe 1f f2 a9 37 3f e1 c3 4c fb cb 18 26
0000130 2e 35 cb 55 d6 e6 06 db 5c 98
```

Posted 5 years ago by [eskimo](#)  

Thanks Eskimo.

I did not verify this using openssl but I did verify it using a java program so I kind of doubt that it will not work using openssl. In my case I must have messed up with the public key by running this program to often at not deleting the public key placed in keystore or maybe one other suspect which is the way you do hashing for small data. I started out using.

```
CC_SHA1_Init(&SHA1);

CC_SHA1_Update(&SHA1, data , usedLen);

unsigned char digest[CC_SHA1_DIGEST_LENGTH];

memset(digest, 0, CC_SHA1_DIGEST_LENGTH);

CC_SHA1_Final(digest, &SHA1);
```


And substituted this to

```
CC_SHA1(data,usedLen,digest);
```

I kind of doubt if this was the problem, but since I am not that convinced that I actually messed up with the public key (since I do believe to have verified the signature with both the generated and stored key) either I thought to mention it for some body else on the verge of a deep depression.

So given that you follow Eskimo's procedure it does work and as you can see I am using Objective C while Eskimo has done this in Swift.

One other strange thing I experienced was that the private key was only good for one go..If you intend to use it again you must retrieve it again. Failing to do so will produce a strange delay and a strange error message about credentials.

Posted 5 years ago by [Ronnyk](#) 

So given that you follow Eskimo's procedure it does work ...



Yay!

One other strange thing I experienced was that the private key was only good for one go..If you intend to use it again you must retrieve it again. Failing to do so will produce a strange delay and a strange error message about credentials.

Well, that's weird. I suspect it's related to the Touch ID integration. Please file a [bug](#) about this, then post your bug number, just for the record.

Share and Enjoy — Quinn "The Eskimo!" Apple Developer Relations, Developer Technical Support, Core OS/Hardware

```
let myEmail = "eskimo" + "1" + "@apple.com"
```

Posted 5 years ago by [eskimo](#)  

< Page 2 of 3 > Last

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the Apple Developer Forums Participation Agreement.

Apple Developer Forums

Discover

macOS
iOS
watchOS
tvOS
Safari and Web
Games
Business
Education
WWDC

Design

Human Interface Guidelines
Resources
Videos
Apple Design Awards
Fonts
Accessibility
Internationalization
Accessories

Develop

Xcode
Swift
Swift Playgrounds
TestFlight
Documentation
Videos
Downloads

Distribute

Developer Program
App Store
App Review
Mac Software
Apps for Business
Safari Extensions
Marketing Resources
Trademark Licensing

Support

Articles
Developer Forums
Feedback & Bug Reporting
System Status
Contact Us

Account
Certificates, Identifiers & Profiles
App Store Connect

To view the latest developer news, visit [News and Updates](#).