2014-02-14.sagews

February 14, 2014

Contents

Video: `http://youtu.be/qtPhffaX_cU`

1 Lecture on Feb 14, 2014 Elliptic Curve Digital Signature Algorithm (ECDSA)

1.1 A very simple ECDSA implementation demo and test

Setup: Choose a prime number q, an elliptic curve E mod q, a point G of some prime order p, and define a set-theoretic map (in any way) $\phi : \mathbf{F}_q \to \mathbf{F}_p^*$. Choose a random secret $d \in \mathbf{F}_p^*$ and let $Q = dG$. The public key is (E, G, Q, p) and the private key is d.

```
q = next_prime(2^128); q
```

```
340282366920938463463374607431768211507
```

```
Fq = GF(q)       # Galois Field: {0,1,2,...,q-1}  work mod q
E = EllipticCurve(Fq, [3,4])   # y^2=x^3+3*x+4
E.cardinality().factor()
```

```
2^2 * 5 * 17 * 1000830490943936657180023443038782793
```

```
E.random_point??
```

```
   File: /usr/local/sage/sage-5.12/local/lib/python2.7/site-
packages/sage/schemes/elliptic_curves/ell_finite_field.py
   Source:
       def random_element(self):
        """
        Returns a random point on this elliptic curve.
```

```
        If `q` is small, finds all points and returns one at random.
        Otherwise, returns the point at infinity with probability
        `1/(q+1)` where the base field has cardinality `q`, and then
        picks random `x`-coordinates from the base field until one
        gives a rational point.

        EXAMPLES::

            sage: k = GF(next_prime(7^5))
            sage: E = EllipticCurve(k,[2,4])
            sage: P = E.random_element(); P
            (16740 : 12486 : 1)
            sage: type(P)
            <class
'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
            sage: P in E
            True

        ::

            sage: k.<a> = GF(7^5)
            sage: E = EllipticCurve(k,[2,4])
            sage: P = E.random_element(); P
            (2*a^4 + 3*a^2 + 4*a : 3*a^4 + 6*a^2 + 5 : 1)
            sage: type(P)
            <class
'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
            sage: P in E
            True

        ::

            sage: k.<a> = GF(2^5)
            sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
            sage: P = E.random_element(); P
            (a^4 + a^2 + 1 : a^3 + a : 1)
            sage: type(P)
            <class
'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
            sage: P in E
            True

        Ensure that the entire point set is reachable::

            sage: E = EllipticCurve(GF(11), [2,1])
            sage: len(set(E.random_element() for _ in range(100)))
            16
            sage: E.cardinality()
            16

        TESTS:
```

```
        See trac #8311::

            sage: E = EllipticCurve(GF(3), [0,0,0,2,2])
            sage: E.random_element()
            (0 : 1 : 0)
            sage: E.cardinality()
            1

            sage: E = EllipticCurve(GF(2), [0,0,1,1,1])
            sage: E.random_point()
            (0 : 1 : 0)
            sage: E.cardinality()
            1

            sage: F.<a> = GF(4)
            sage: E = EllipticCurve(F, [0, 0, 1, 0, a])
            sage: E.random_point()
            (0 : 1 : 0)
            sage: E.cardinality()
            1

        """
        random = current_randstate().c_rand_double
        k = self.base_field()
        q = k.order()

        # For small fields we find all the rational points and pick
        # one at random.  Note that the group can be trivial for
        # q=2,3,4 only (see #8311) so these cases need special
        # treatment.

        if q < 5:
            pts = self.points() # will be cached
            return pts[ZZ.random_element(len(pts))]


        # The following allows the origin self(0) to be picked
        if random() <= 1/float(q+1):
            return self(0)

        while True:
            v = self.lift_x(k.random_element(), all=True)
            if v:
                return v[int(random() * len(v))]
```

```
# Found via -- P = E.random_point(); P
P = E([28164262154109634856772136899605249 3558, \
   321403994476306244071060762777806837 85])
```

```
factor(P.order())
```

2 * 100083049094393665718002344303878279 3

```
G = 2*P; p = G.order(); p
Fp = GF(p)
```

```
100083049094393665718002344303878279
```

```
G
```

```
(24396559400458357354668041023631381647 : 28533677569567554224304596712465927572 : 1)
```

```
def phi(x):
    a = Fp(x.lift())
    if a == 0:
        a = Fp(1)
    return a
```

```
#d = Fp.random_element()
d = Fp(85509169948493851489056561321083269)
print "secret =", d
```

```
secret = 85509169948493851489056561321083269
```

```
Q = lift(d)*G    # lift(d) is integer in [0..p-1]
```

```
public_key = {'E':E, 'G':G, 'Q':Q, 'p':p}
```

```
public_key
```

```
{'Q': (12483616377792891950229773057537048628 : 22935360844831174748186076932151161339 :
1), 'p': 100083049094393665718002344303878279, 'E': Elliptic Curve defined by y^2 = x^3 +
3*x + 4 over Finite Field of size 340282366920938463463374607431768211507, 'G':
(24396559400458357354668041023631381647 : 28533677569567554224304596712465927572 : 1)}
```

Lets sign something

Hash: Hash the message m to an element $z \in \mathbf{F}_p^*$.

```
message = "This is math 480.  It's a very flexible class about various \
    things. -- William"
import hashlib
h = hashlib.sha1(message).hexdigest(); h
```

```
'e77f52876de8572f187bb226479f7268ec9464d0'
```

```
%timeit  hashlib.sha1(message).hexdigest()
```

```
625 loops, best of 3: 1.4 s per loop
```

```
%timeit hash(h)
```

```
625 loops, best of 3: 92.7 ns per loop
```

```
# But we need a number modulo p, so
z = hash(h) % p; z
```

```
4318374665117912394
```

Random Point: Choose a random $k \in \mathbf{F}_p^*$, and compute $kG \in E(\mathbf{F}_q)$.

```
k = Fp.random_element()
print "k =",k
kG = lift(k)*G
```

```
print "kG = ",kG
```
```
k = 23400802509337484411241379049672603 8
kG =  (91683268263023261246596732651133609132 : 15830128093980840535716800166550952813 8 :
1)
```

Compute Signature: Compute

$$r = \phi(x(k(G))) \in \mathbf{F}_p^* \quad \text{and} \quad s = \frac{z + rd}{k} \in \mathbf{F}_p.$$

```
r = phi(kG[0]); s = (z+r*d)/k
sig = (r,s)
print "sig =", sig
```
```
sig = (60769358712502544321459933460437496 9, 18866187416561819133606403539278673 8)
```

Hash: Hash message m to the same $z \in \mathbf{F}_p^*$ as above.

Verify: Compute the point

$$C = \frac{z}{s}G + \frac{r}{s}Q \in E(\mathbf{F}_q).$$

The signature is valid if $\phi(x(C)) = r$.

```
z
```
```
4318374665117912394
```

```
C = lift(z/s)*G + lift(r/s)*Q; C
```
```
(91683268263023261246596732651133609132 : 15830128093980840535716800166550952813 8 : 1)
```

```
phi(C[0]) #  == r  ? yep
```
```
60769358712502544321459933460437496 9
```

Prop: If (r, s) is valid, then this protocol concludes it is valid.

Proof: Since (r, s) is valid, we have $s = (z + rd)/k$, so $k = (z + rd)/s$. Thus

$$kG = \frac{z}{s}G + \frac{rd}{s}G = \frac{z}{s}G + \frac{r}{s}Q = C.$$

Lets sign another document

```
message2 = "This is a very flexible class about various things. -- \
   William"
h2 = hashlib.sha1(message2).hexdigest()
z2 = hash(h2) % p
r2 = phi(kG[0]); s2 = (z2+r2*d)/k
sig2 = (r2, s2)
print "sig2 =", sig2
```
```
sig2 = (60769358712502544321459933460437496 9, 34203988539338435147022284130019108 6)
```

And verify the signature

```
C2 = lift(z2/s2)*G + lift(r2/s2)*Q; C2
```

(91683268263023261246596732651133609132 : 158301280939808405357168001665509528138 : 1)

```
phi(C2[0])              # == r2 above.
```

607693587125025443214599334604374969

Question: What serious mistake did we just make?!

```
# Just looking at the signatures, we can easily compute this number:
print (z - z2)/(s-s2)

# Wait, that's actually k, which was some secret thing used in signing...\
    so?
k
```

234008025093374844112413790496726038
234008025093374844112413790496726038

```
# so!
(s*k-z)/r   # all known by attacker
```

85509169948493851489056561321083269

```
# Umh, that's the private key. Crap.
d
```

85509169948493851489056561321083269

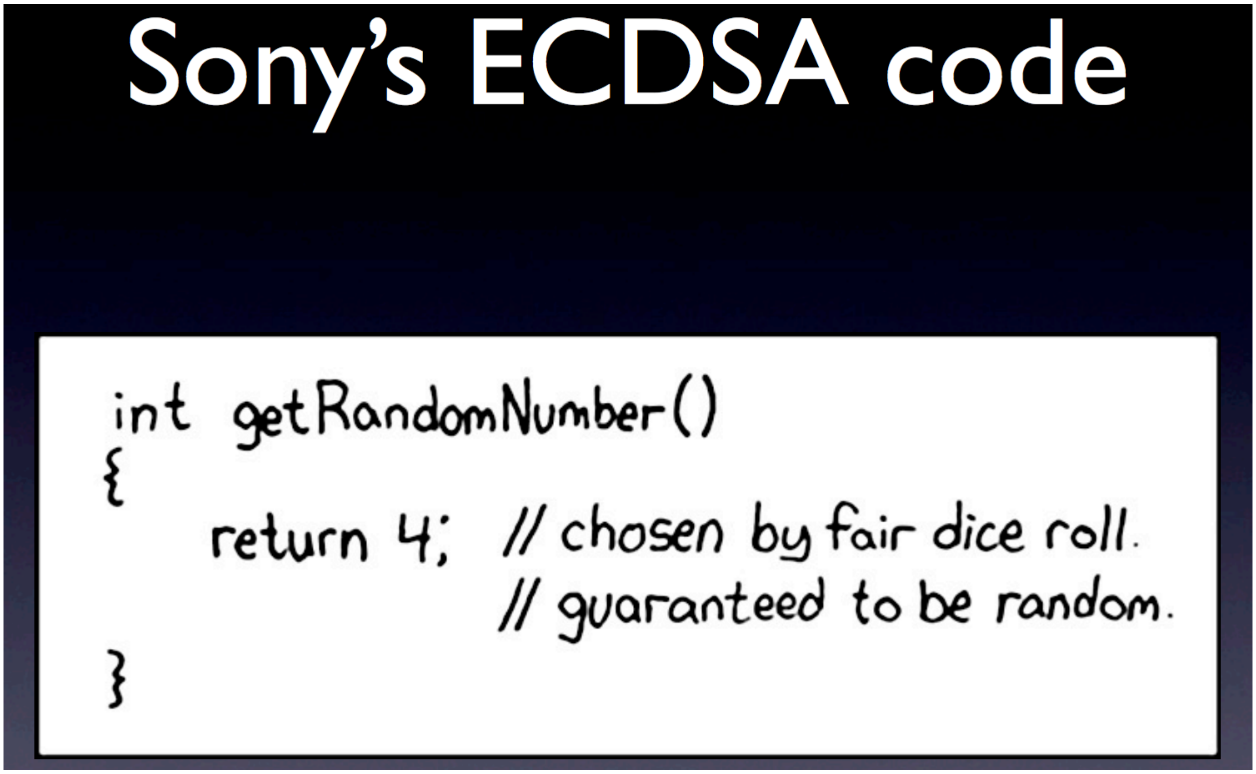
1.1.1 Our mistake

Our mistake was that we didnt generate a new random k. In general, if the ks arent really damn random ECDSA will be easily crackable.

1.2 ECDSA in PS3 an egrarious example

They used one single k, not changing it at all, leading to them being totally owned.


```
salvus.file('ps3-random.png')
```



http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf

1.2.1 ECDSA in PS3 has since been fixed

Anyone still workin on the ECDSA code?

Question is in the title!

But seriously though. Is anyone working on the ECDSA code/private keys still,or has everyone given up and stuck to what they know?

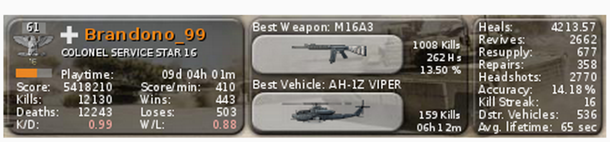
I have a strange idea on how to get the keys/core files...

Could we not disassemble a PS3 motherboard very carefully and power individual components? Basically, couldnt we just dissect and isolate the ps3's guts,and then work from there?

I hope you guys answer my questions,

Bye for now!
(p.s if you ever read this Hellsing,show more pictures of your lovely b00biez :D)

*Last edited by Brandono_99; 03-05-2013 at 04:48 PM. **Reason:** Not enough manliness involved.*



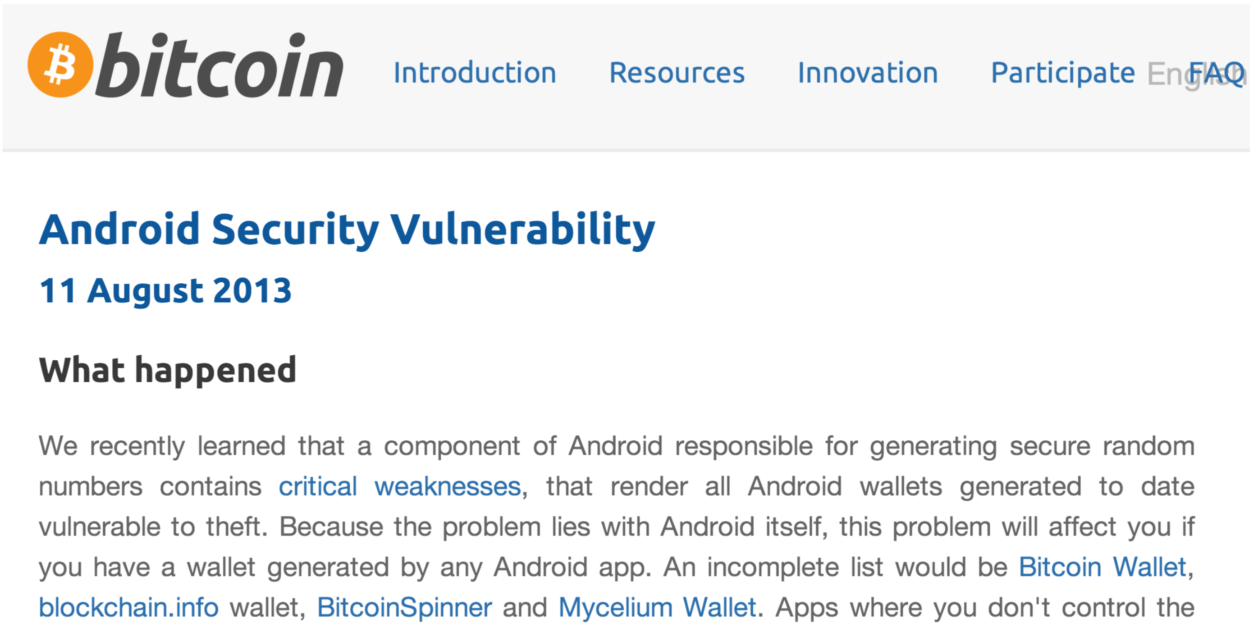
Reply With

the private keys are not in the ps3 !!!! they are in sony's office

1.3 ECDSA in Bitcoin

Yes, people have messed up the implementation of ECDSA here too, leading to theft

```
salvus.file('bitcoin-random.png')
```




```
Very briefly the built in "SecureRandom" Java function in all Android
phones had a serious bug in it, which meant basically all crypto ever
deployed for years in these phones was potentially broken.  Since
people use bitcoin on Android, they were impacted, and there were
actual exploits of this.   Somebody described the bug thus: " The
problem happens when creating a self seeding instance of SecureRandom
(i.e., no seed, either through the constructor or through setSeed
method, is passed by the programmer). The seed is stored in a buffer
with the seed data, a counter, and padding. In the case where no seed
is passed by the programmer, a bug in the code caused a pointer into
the buffer to not be updated, which causes other code to overwrite
portions of the seed.... The result is that there is only 64 bits of
entropy in the buffer. This is much, much too low."

See http://crypto.stackexchange.com/questions/9694/technical-details-of-\
    attack-on-android-bitcoin-usage-of-securerandom
```

1.4 The Bitcoin Elliptic Curve

Definition: https://en.bitcoin.it/wiki/Secp256k1

Discussion: https://bitcointalk.org/?topic=2699.0

ECDSA verification is the primary CPU bottleneck for running a network node. So if Koblitz curves do indeed perform better we might end up grateful for that in future

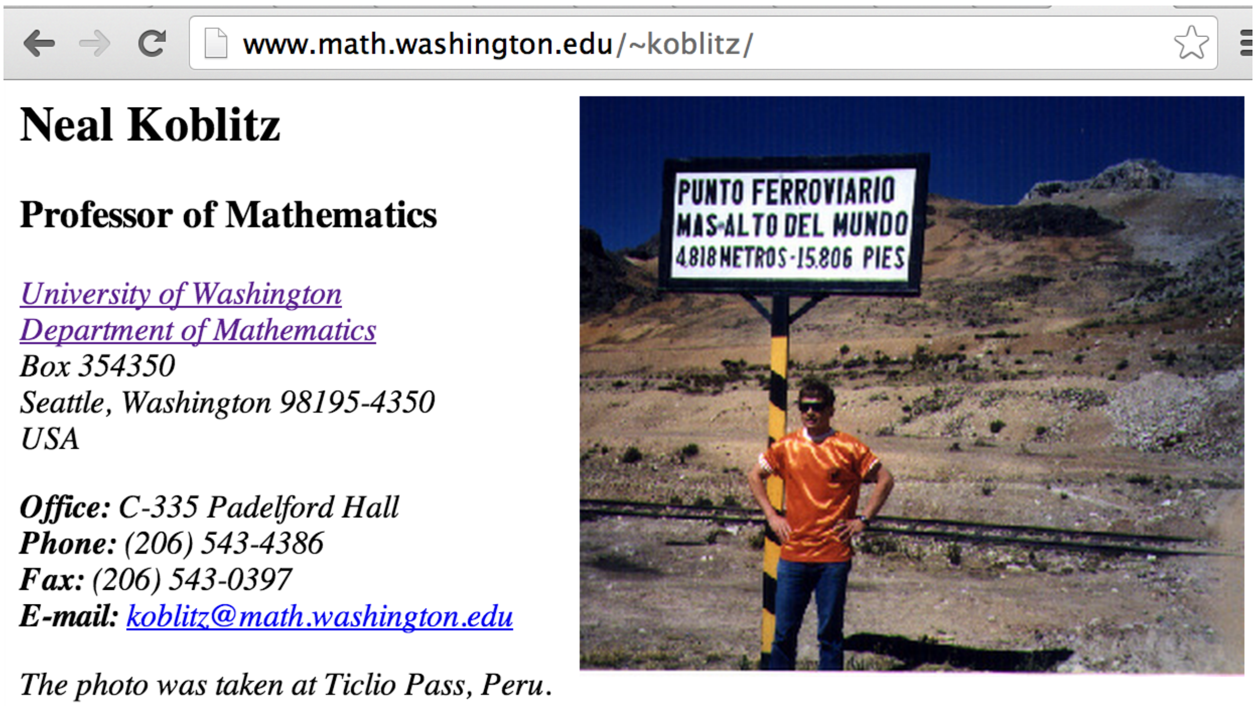
```
q =  2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
```

```
is_prime(q)
q
```

```
True
115792089237316195423570985008687907853269984665640564039457584007908834671663
```

```
# This is the elliptic curve "Secp256k1", where the "k" stands for "\
    Koblitz".
E = EllipticCurve(GF(q),[0,7]); E
```

```
Elliptic Curve defined by y^2 = x^3 + 7 over Finite Field of size
115792089237316195423570985008687907853269984665640564039457584007908834671663
```




```
%time p = E.cardinality(); p
```

```
115792089237316195423570985008687907852837564279074904382605163141518161494337
CPU time: 0.01 s, Wall time: 0.01 s
```

```
is_prime(p)
```

```
True
```

```
len(p.str(2))
```

```
256
```

```
s = '79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16\
    F81798'.replace(' ','').lower(); s
```

```
'79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798'
```

```
x = E.base_field()(ZZ(s,base=16)); x
```

```
55066263022277343669578718895168534326250603453777594175500187360389116729240
```

```
G = E.lift_x(x)
```

```
G
-G
```

```
(55066263022277343669578718895168534326250603453777594175500187360389116729240 :
32670510020758816978083085130507043184471273380659243275938904335757337482424 : 1)
(55066263022277343669578718895168534326250603453777594175500187360389116729240 :
83121579216557378445487899878180864668798711284981320763518679672151497189239 : 1)
```

```
ZZ(G[1]).str(base=16)  # this is the one
ZZ(-G[1]).str(base=16)
```

```
'483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8'
'b7c52588d95c3b9aa25b0403f1eef75702e84bb7597aabe663b82f6f04ef2777'
```

```
G
```

```
(55066263022277343669578718895168534326250603453777594175500187360389116729240 :
32670510020758816978083085130507043184471273380659243275938904335757337482424 : 1)
```

```
G.order()
```

```
115792089237316195423570985008687907852837564279074904382605163141518161494337
```