TJCTF 2015: Curvature Writeup

May 8, 2015 - Peter Heppenstall

The Problem

We are given a server that acts as an oracle doing elliptic curve scalar point multiplication (ECSPM) with a given point. Using this oracle we must solve the elliptic curve discrete logarithm problem (ECDLP) for a constant k, which is our flag.

Finding the Vulnerability

Checking out the source code we are given the coefficients a and b, as well as the finite field $\mathbb{F}_\mathrm{p}$ the elliptic curve is defined over.

```
A = 0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C126DC5C6CE94A4B44F330B5D9
B = 0x26DC5C6CE94A4B44F330B5D9BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6
P = 0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377
# Curve is y^2 = x^3 + Ax + B, all modulo P
```

If this curve looks a little suspicious it's because it happens to be BrainpoolP256t1, a somewhat-rigid twisted curve, which has some overlap in its coefficients:

A = 0x7D5A0975FC2C3057EEF67530417AFFE7FB8055C1**26DC5C6CE94A4B44F330B5D9**
B = 0x**26DC5C6CE94A4B44F330B5D9**BBD77CBF958416295CF7E1CE6BCCDC18FF8C07B6

Unfortunately while certainly suspicious it doesn't seem to make the curve cryptographically insecure.

*(**Technically this curve has a cryptographically low cost ($2^{44.5}$) for a combined attack using low order points and could also probably be exploited, but there is an easier and less expensive method.**)*

With no inherent weakness in the curve itself we need to check for insecurities elsewhere.

Reading through the code we can find that it does the ECSPM with custom functions `multiply()`, `double()`, and `add()`, and that they were used in the function `handle()`:

```
# Point Addition
def add(a, b):
    if a == 0:
        return b
    if b == 0:
        return a
    l = ((b[1] - a[1]) * inv(b[0] - a[0])) % P
    x = (l*l - a[0] - b[0]) % P
    y = (l*(a[0] - x) - a[1]) % P
    return (x,y)


#Point Doubling
```

```
def double(a):
    if a == 0:
        return a
    l = ((3*a[0]*a[0] + A) * inv(2*a[1])) % P
    x = (l*l - 2*a[0]) % P
    y = (l*(a[0] - x) - a[1]) % P
    return (x,y)

#Scalar Point Multiplication
def multiply(point, exponent):
    # No timing attack for you :P
    r0 = 0
    r1 = point
    for i in bin(exponent)[2:]:
        if i == '0':
            r1 = add(r0, r1)
            r0 = double(r0)
        else:
            r0 = add(r0, r1)
            r1 = double(r1)
    return r0

class ThreadedServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    allow_reuse_address = True

class Handler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.request.send("Send me points to exponentiate!\n")
        self.request.send("Format: \"x y\"\n")

        while True:
            point = self.request.recv(4096)
            x, y = [int(i) for i in point.strip().split()]
            x2, y2 = multiply((x, y), FLAG)
            self.request.send("Your point:\n")
            self.request.send("(%d, %d)\n" % (x2, y2))
```

Reading through we can see that the script takes a point G and using the curve $\mathbb{E}\colon \quad y^2 = x^3 + Ax^2 + B$ where $A, B \in \mathbb{F}_\text{p}$ computes and returns $G' = kG$. Checking the server's math functions we can see that mathematically they work, but they don't use the point B in their calculations! This is the standard for short Weierstrass curves like the one the server is using, however since the server also neglets to verify $G \in E(\mathbb{F}_\text{p})$ or $G' \in E(\mathbb{F}_\text{p})$ the server is vulnerable to an invalid-curve attack which we can use to recover k.

The Vulnerability

Since the server doesn't use B in its calculations we can define a new curve using a coefficient C instead of B where $C \neq B$ in our short Weierstrass equation from earlier. Let's call this new curve $\mathbb{E}'$.

$$\mathbb{E}'\colon \quad y^2 \equiv x^3 + Ax^2 + C \pmod{P}$$

$$C \neq B$$

Choosing C such that this new curve's order has a small prime divisor r will ensure there exists a subgroup of $\mathbb{E}'$ of small order r. We can then send the server a point of this subgroup, and since neither C nor B is used by the server the ECSPM will actually occur over our new curve $\mathbb{E}'$. As the discrete logarithm is now in the subgroup of $|r|$ generated by $G \in \mathbb{E}'$, the result will only have r possible values! Since there is such a small keyspace and the multiplicative group's order is a smooth integer, we can then easily solve the ECDLP using the Pohlig-Hellman algorithm. This will leak $k \mod r$. Repeating this for new curves and new r values will create a system of linear congruences which can be solved with the Chinese Remainder Theorem, thus yielding the final value of k.

$$|\mathbb{E}'| \mod r = 0$$

$$|G| = r$$

$$G' = kG$$

$$X = k \mod |G|$$

$$k \equiv X \pmod r$$

Implementing and Executing the Vulnerability

Now comes the fun part, writing all this up in a Sage Script. Sage is crucial here as it will do all of the ECC work for us, as well as letting us communicate with the server and do logic in Python. Technically a script isn't necessary as we could do everything by hand in the Sage terminal, however a script saves time and lowers the likelihood of error. This is especially useful since we will need enough linear congruences to find the correct value of k.

The Code:

Note: The server seems to not like points that are generated from primes that are too small or result in points that are too large. As far as I can tell this is a bug on their end.

```
#!/usr/bin/env sage -python2

import socket
from sage.all import *

#A function to communicate a point with the server
#Probably recv's and sends too many bytes, but I was in a rush
def compute(g):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect(('p.tjctf.org', 8092))
    sock.recv(4096)
    resp = sock.recv(4096).strip('\n')
    sock.sendall(g)
    resp = sock.recv(4096).strip('\n')
    return sock.recv(4096).strip('\n')

#Define the Curve
P = 768849563970453442208097466290016490930379502009430552037356014450315l6
197751
A = 566981876053261100436272283961783460771206145394752141093868281887638 84
139993
B = raw_input("Enter a B value: ")
E = EllipticCurve(GF(P),[A,int(B)])
print prime_factors(E.order())

#Define a prime to leak K with
prime = raw_input("Enter a prime: ")

#Calculate the point to send to the server and save the result
G = E.gen(0) * int(E.order() / int(prime))
G1 = compute(str(G).replace('(', '').replace(' : 1)', '').replace(' : ', '
'))
G1 = E(int(G1[1:G1.index(',')]), int(G1[G1.index(' ')+1:len(G1)-1]))

#Perform the Discrete Log
print "Solving Elliptic Curve Discrete Log With Pollig-Hellman..."
print "K mod "+prime+" = " + str(G.discrete_log(G1))
```

Running this with a few values I arrived at:

$$k = 270537299 \pmod{555738749}$$

$$k = 627965704 \pmod{1342839413}$$

$$k = 59042701956 \pmod{147158786267}$$

$$k = 420403779641 \pmod{453660276361}$$

$$k = 9750737702548 \pmod{12298541720533}$$

$$k = 12053171659052 \pmod{17745451608589}$$

$$k = 34549005323271 \pmod{49305090929629}$$

We can now solve for k in sage:

```
CRT_list([270537299,627965704,59042701956,420403779641,9750737702548,120531
71659052,34549005323271], [555738749,1342839413,147158786267,453660276361,1
2298541720533,17745451608589,49305090929629])
```

Giving our final value for k:

> **54815471905492393585931182194968733201189578373199788873886877824626330137714**

Encoding this integer as hex and decoding that hex as ascii reverses the encoding shown in the redacted sever's code:

```
FLAG = int("REDACTED".encode("hex"),16) # Find me!
```

We arrive at our final flag:

`y0u're_th3_3llipt1c_curv3_mas7tr`