Iteration IV Report

IU BS Project Spring 2020

Leonid Lygin

Contents

1 Intro

- **Student name**: Leonid Lygin.
- **Topic**: "nice" EC param design.
- **Supervisor**: Phillip Braun.
- **Iteration number**: 4.

2 Initial plan

Initial plan was to (quoting):

> Finally do the wikipedia-grade curves, and maybe look into practical applications, where can we really use these params, what can we do.

After further refining the plan in the meeting, we came to a conclusion of doing either the wikipedia-grade curve, or testing out a letsencrypt-like scenario.

That scenario assumes that you, as an attacker, have access to some web-server in the past, and you would want to decrypt TLS traffic towards that server. Then we can hijack the server's `certbot` to use a curve that is somehow flawed, and `letsencrypt` would probably sign the produced certificate.

3 Actual work done

TL;DR: I've tried to implement the letsencrypt way, it didn't work, so there is a generator that outputs curves which are suitable for Smart's attack.

3.1 Letsencrypt

Turns out that it supports only named curves, and even then only some of them, so out of luck here. The error comes from the server, not from the client, so somehow circumventing that is not feasible.

3.2 Smart's attack

When we have a curve over $GF(p)$, if the generator's order is equal to p, the curve is vulnerable to Smart's attack (described in [1], together with an implementation in `sage`).

Generating the curves is another whole topic, which was the main time sink here. The algorithm is described in [2] in great detail, but still, no code. Also, the paper provides example parameters, which are wrong and do not work, so I needed to actually dig in and find the error (maybe the error is intentional, like in some online exploits to ward off script kiddies).

Anyway, the generator is spread across multiple files with `sage` and C code, it is a bit of a mess, but it works.

- `scripts/smart_attack.sage` contains an implementation for the attack itself.

  It expects the elliptic curve params in the arguments in decimal, and spits out what it thinks is the private key.

- `c_tools/bin/crack_ec_smart` contains an wrapper for the above script to work with OpenSSL keys.

  It performs loading, parsing, and serializing, all that dirty work.

- `scripts/generate_p_equal_to_n_params.sage` contains the actual generator of a Smart-vulnerable curve (with hardcoded params, but a rather big table of D and j is in the paper, so changing the key size would not be a problem).

[1] https://wstein.org/edu/2010/414/projects/novotney.pdf
[2] http://www.monnerat.info/publications/anomalous.pdf

- `c_tools/bin/generate_curve_by_params` is a wrapper for the generator script, to work with OpenSSL format.
- `scripts/generate_smart_params.sh` is a convenience script that would generate the curve, and a key pair to go with it.

3.3 Smart's attack in action

Let's get to the actual commands and outputs.

```
$ ./generate_smart_params.sh
<makefile outputs>
===== Generating the params and writing them into the file =====
==================== Validating the params =====================
Field Type: prime-field
Prime:
    00:80:00:00:00:00:00:00:00:22:bf:e7:b0:15:4d:
    1a:5c:dd:d6:19:6b
A:
    4a:91:55:f7:73:69:ea:27:06:78:58:6d:3a:7a:cf:
    3d:39:83:d9:9a
B:
    23:9f:1c:05:b3:0e:b9:3b:68:2f:b4:d7:3c:8c:32:
    15:18:36:d5:36
Generator (uncompressed):
    04:02:c5:24:a1:66:2d:e8:6a:ba:11:30:29:f1:a4:
    6b:88:a6:bc:1a:32:71:fb:65:b8:36:07:e0:d1:6e:
    90:13:68:a7:7d:4c:a0:02:a8:ec:73
Order:
    00:80:00:00:00:00:00:00:00:22:bf:e7:b0:15:4d:
    1a:5c:dd:d6:19:6b
Cofactor:  1 (0x1)
=================== Generating a private key ===================
====================== Validating the key ======================
read EC key
Private-Key: (160 bit)
priv:
    28:07:b4:08:49:27:63:65:34:6f:af:71:e9:5e:b9:
    87:39:50:ac:9e
pub:
    04:6b:b5:50:9c:d8:84:24:89:ee:8f:cc:e5:d7:db:
    13:be:9d:06:e8:84:49:13:c8:67:0a:5d:fd:3b:74:
    c1:73:fa:a0:2f:9a:b9:4b:4c:a4:a9
<same params as above>
=================== Generating a public key ====================
read EC key
writing EC key
```

```
================== Validating the public key ===================
read EC key
Private-Key: (160 bit)
pub:
    04:6b:b5:50:9c:d8:84:24:89:ee:8f:cc:e5:d7:db:
    13:be:9d:06:e8:84:49:13:c8:67:0a:5d:fd:3b:74:
    c1:73:fa:a0:2f:9a:b9:4b:4c:a4:a9
<same params as above>
```

We can see here that the private key is relatively random, at least compared to the previous degenerate curve, where there were only 3 keys to choose from.

Also, we can notice that the order is equal to the prime, so Smart's attack is possible.

So, let's try to break this!

```
$ c_tools/bin/crack_ec_smart \
    files/smart_key.pub.pem \
    files/smart_cracked_key.pem
$ openssl ec -in \
    files/smart_cracked_key.pem -text -noout
read EC key
Private-Key: (160 bit)
priv:
    28:07:b4:08:49:27:63:65:34:6f:af:71:e9:5e:b9:
    87:39:50:ac:9e
pub:
    04:6b:b5:50:9c:d8:84:24:89:ee:8f:cc:e5:d7:db:
    13:be:9d:06:e8:84:49:13:c8:67:0a:5d:fd:3b:74:
    c1:73:fa:a0:2f:9a:b9:4b:4c:a4:a9
<same params>
```

As we can see, the private key is really the same as above, and the crack only took us 2 seconds!

4 References to results

All results can be found in the github repo: https://github.com/ionagamed/iu-bs3-project.

5 Current issues

Letsencrypt didn't work, and we still probably need to somehow sign the certificate for general use, so I'm looking into domain registrars that would allow us to sign a custom certificate with custom crypto, but that doesn't look promising.

Also, there are much cooler attacks, which are harder to detect, so implementing them would be nice.

6 Planned for the next iteration

Again, a combined plan - research domain registrars which would allow custom crypto in certificates, and implement different harder to detect curves (now that I have a somewhat streamlined approach of doing the math with `sage` and encoding with C).