

Challenge SSTIC 2026 - Step 3 : lobster128 parameters + overflowing faults

nebucca

Analyst,

Thanks for unlocking access. It seems to be far worse than we feared. As you may have read, SAFE does not respond to commands. After a lengthy discussion with one of Aegis admins, we discover the existence of an undocumented, and quite frankly, illegal "debug monitoring stream". In fact, a webcam is pointed at a screen connected to SAFE system (also undocumented...).

However, using this *monitoring stream*, we were able to identify the problem : SAFE's signing key have been updated by an unknown party (we stongly suspect the hactivist group). Can you investigate :

How the attacking group was able to break SAFE's signature scheme. It was supposed to be state-of-the-art PQC. If there is a flaw, especially one that can be found by hacktivist nobodies, we have to address it immediatly. If the flaw used by the attacking group cannot be reused for any reasons, can you look at their signing scheme ?

It is referred to as lobster256 in their documentation, and we hadn't found a public match. We cannot rule out a custom algorithm, and so far, Aegis have proven to be lacking in their security. If we were able to fake signatures, we would be able to send an update key packet and gain a foothold in the SAFE system. We had found a partial backup in one of Aegis employee

As two is better than one, and without diminishing your cryptanalysis expertise, we have asked an internal expert to review lobster256 implementation. They are still working on it, and it may take a few days. We will contact you later using the usual channel.

-ñ

1 Reconnaissance

Comme élément nous récupérons un extrait audio plutôt divertissant dans lequel on découvre la genèse du projet lobster.

Nous obtenons aussi une archive du projet contenant :

- le binaire du projet *lobster256*,
- les sources de la bibliothèque *lobster_ECC* utilisée par le projet,
- deux scripts Sage,
- différents articles concernant l'ECC.

La bande audio indique que la bibliothèque est basée sur la *libECC*. Après un diff entre *lobster_ECC* et la *libECC*, on constate que le script *lobster256.sage* reprend fidèlement ce qui a été fait dans *lobster_ECC*.

Suite à la Step précédente, nous savons comment créer des fichiers qui sont transmis à *diode_dest*.

Nos premiers tests provoquent une erreur qui nous apprend que la signature doit faire 88 bytes.

On adapte donc notre signature à la bonne taille.

Nouveau message d'erreur indiquant que la signature n'a pas un format base64 valide.

Nouveau fichier avec 88 'A' pour signature.

Message d'erreur indiquant que la clef publique est corrompue.

2 Que fait *lobster256* ?

Lorsqu'on lance *lobster256*, on voit que plusieurs actions sont possible : *gen_keys*, *sign*, *verify*.

La commande :

```
./lobster256 verify
```

nous apprend que :

```
arg1 = input file to verify
arg2 = input file containing the ignition parameters (in raw
      binary format)
arg3 = input file containing the public key (in raw binary
      format)
arg4 = input file containing the signature
```

Nous connaissons la signature que nous voulons vérifier. Nous connaissons la clef publique utilisée. Mais que contient le fichier ignition ?

Grâce à *Ghidra* on apprend que le contenu du fichier doit commencer par "IGNITION" et qu'il contient les valeurs de a et b afin de "configurer" la courbe elliptique utilisée par l'algorithme de signature :

$$y^2 = x^3 + ax + b \quad (E)$$

```
if (((char)lenread == 'H') &&
    (rsltdecode = memcmp(ignitionData,&IGNITION_MAGIC,8), rsltdecode
    == 0)) {
    a.3._0_8_ = ignitionData._8_8_;
    a.3._8_8_ = ignitionData._16_8_;
    a.3._16_8_ = ignitionData._24_8_;
    a.3._24_8_ = ignitionData._32_8_;
    b.2._0_8_ = ignitionData._40_8_;
    b.2._8_8_ = ignitionData._48_8_;
    b.2._16_8_ = ignitionData._56_8_;
    b.2._24_8_ = ignitionData._64_8_;
    rsltdecode = import_params(ctx,ec_s_parms,a_str_param.1,
    b_str_param.0);
```

3 Fichier IGNITION : Déterminer a et b

L'implémentation des opérations sur les courbes elliptiques semble équivalente entre *lobster256* et le script Sage. Aussi, on va mettre de côté le binaire pour le moment.

La bande audio semble indiquer qu'il faudrait creuser du côté des valeurs de G , $2G$, $3G$, $4G$, $5G$, $6G$, $7G$ stockées dans *COMP_WIN*. Il s'agit de la représentation compacte des points. Nous ne connaissons que les x .

Par définition de l'opération d'addition nous savons que :

$$P = (x_1, y_1), \quad Q = (x_2, y_2)$$

si P et Q sont distincts :

$$\begin{aligned}\lambda &= \frac{y_2 - y_1}{x_2 - x_1} \\ x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1\end{aligned}$$

si P et Q sont identiques (cas du doublement d'un point) :

$$\begin{aligned}\lambda &= \frac{3x_1^2 + a}{2y_1} \\ x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1\end{aligned}$$

Considérons :

$$G = (x_1, y_1), \quad 2G = (x_2, y_2), \quad 3G = (x_3, y_3)$$

On a :

$$2G = G + G$$

d'où :

$$\begin{aligned}y_2 &= \lambda_1(x_1 - x_2) - y_1 \\ y_2 + y_1 &= \lambda_1(x_1 - x_2) \\ \lambda_1(x_1 - x_2) &= (y_2 + y_1) \\ \lambda_1^2(x_1 - x_2)^2 &= (y_2 + y_1)^2 \\ \lambda_1^2(x_1 - x_2)^2 &= y_2^2 + y_1^2 + 2y_1y_2\end{aligned}$$

Et :

$$3G = 2G + G$$

d'où :

$$\begin{aligned}\lambda_2 &= \frac{y_2 - y_1}{x_2 - x_1} \\ \lambda_2(x_2 - x_1) &= (y_2 - y_1) \\ \lambda_2^2(x_2 - x_1)^2 &= (y_2 - y_1)^2 \\ \lambda_2^2(x_2 - x_1)^2 &= y_2^2 + y_1^2 - 2y_1y_2\end{aligned}$$

En combinant les deux équations :

$$\lambda_1^2(x_1 - x_2)^2 + \lambda_2^2(x_2 - x_1)^2 = 2y_2^2 + 2y_1^2$$

G et 2G appartiennent à la courbe, d'où :

$$\begin{aligned}y_1^2 &= x_1^3 + ax_1 + b \\ y_2^2 &= x_2^3 + ax_2 + b\end{aligned}$$

On obtient ainsi :

$$\lambda_1^2(x_1 - x_2)^2 + \lambda_2^2(x_2 - x_1)^2 = 2(x_1^3 + ax_1 + b) + 2(x_2^3 + ax_2 + b)$$

Autrement dit :

$$\lambda_1^2(x_1 - x_2)^2 + \lambda_2^2(x_2 - x_1)^2 - 2x_1^3 - 2x_2^3 = (2x_1 + 2x_2)a + 4b$$

$$\boxed{\frac{\lambda_1^2(x_1 - x_2)^2 + \lambda_2^2(x_2 - x_1)^2 - 2x_1^3 - 2x_2^3}{4} = \frac{(x_1 + x_2)}{2}a + b}$$

On connaît :

$$x_1, x_2, x_3$$

et :

$$\lambda_1^2 = x_1 + x_2 + x_3$$

$$\lambda_2^2 = 2x_1 + x_3$$

On a donc une équation linéaire à 2 inconnues.

La même relation lie $2G$, $4G$ et $6G$.

Nous avons donc un système linéaire à 2 équations et 2 inconnues (cf *Solve-ab.sage*).

On trouve ainsi a et b .

On reconstruit alors le fichier de paramétrage de la courbe (cf *ignitionFile.py*).

On a maintenant tous les éléments pour vérifier une signature.

4 La clef publique est corrompue

On lance le debugger *edb* pour regarder plus en détails le message d'erreur indiquant que la clef publique est corrompue.

On reproduit assez facilement le problème : la signature qui fait 64 octets de long est encodée en base 64.

En base 64, chaque caractère encode 6 bits. Aussi pour encoder des octets, on utilise 4 caractères pour encoder 3 octets.

Pour encoder la signature de 64 octets, il faut donc une chaîne de 88 caractères (multiple de 4). La chaîne peut encoder 2 octets de plus que nécessaire pour la signature.

Or, l'espace prévu sur la stack pour cette variable est exactement de 64 octets. Ainsi, si on crée une signature qui se termine par autre chose que du padding, il est possible de déborder sur la variable contiguë et il s'avère qu'il s'agit de la clef publique.

Le premier octet a un rôle technique et ne doit pas être modifié, mais le deuxième correspond à l'octet de poids faible du x de la clef publique.

Si nous voulons pouvoir passer la vérification de la signature, la seule solution semble de forger une signature. Retrouver la clef privée semble impossible (pas d'accès à des opérations impliquant la clef privée et pas d'indices pour la régénérer). Cela tombe bien, parmi les articles présents dans le projet, il y a : *Fault Attacks on ECC Signature Verification*.

5 ECKSDSA

L'algorithme utilisé pour la vérification de la signature consiste à :

$$\begin{aligned}
 m &= \text{message} \\
 n &= \text{ordre}(E) \\
 (r, s) &= \text{signature} \\
 h &= \text{hash}(x_{\text{pub}} + y_{\text{pub}} + m) \\
 e &= (r \oplus h) \bmod n \\
 S &= e \times G \\
 X &= s \times \text{Pub} \\
 W &= S + X \\
 r &= \text{hash}(x_W)
 \end{aligned}$$

On part du principe que les fonctions de hashage ne sont pas réversibles. Ainsi il n'est pas possible de trouver un x_W à partir de la valeur de r .

Par conséquent, x_W est fixé, donc r est fixé et au final, trouver une signature pour un message m donné revient à trouver s tel que :

$$W - e \times G = s \times \text{Pub}$$

Il s'agit donc de résoudre le problème du logarithme discret (ECDLP).

La courbe elliptique utilisée ne semble pas présenter de fragilités particulières mais nous savons que nous pouvons modifier légèrement la valeur de X_{pub} .

Cette légère modification a pour effet de pousser le point correspondant à la clef publique en dehors de la courbe utilisée.

Mais comment vont se comporter les calculs face à ce point ?

5.1 Calcul de : $X = s \times \text{Pub}$

La multiplication est calculée grâce à :

```

def XZ_EXP(k, xP, yP, a, b, p):
    (X_R0, Z_R0) = (K(xP), K(1))
    (X_R1, Z_R1) = XZ_DBL(K(X_R0), K(Z_R0), a, b, p)
    kb=Integer(k)
    nb=kb.nbits()
    for i in reversed(range(nb-1)):
        bit=(kb>>i)&1
        if bit==0:
            (X_R1, Z_R1) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R0, Z_R0) = XZ_DBL(X_R0, Z_R0, a, b, p)
        else:
            (X_R0, Z_R0) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R1, Z_R1) = XZ_DBL(X_R1, Z_R1, a, b, p)
    xR0 = X_R0 * ((Z_R0)**(-1))
    xR1 = X_R1 * ((Z_R1)**(-1))
    # Marc Joye's formula : "Weierstass Elliptic Curves and Side-Channel
    Attacks" (8)
    yR0 = (2*b + (a + xP * xR0) * (xP + xR0) - xR1 * (xP - xR0)**2) * (2
        * (yP))**(-1)
    print(f"-->␣(xR0,␣yR0)␣=␣({hex(xR0)},␣{hex(yR0)})")
    return (xR0, yR0)

```

La multiplication du point P n'utilise que x_P , a , b . La valeur de y_P n'est calculée qu'à la fin grâce à la formule de Marc Joye.

Ainsi, le calcul du X résultant reste valide si on considère un point P appartenant à une courbe :

$$dy^2 = x^3 + ax + b \quad (F)$$

Par contre la formule de Marc Joye n'est plus valide car elle repose sur une courbe de la forme :

$$y^2 = x^3 + ax + b \quad (E)$$

5.1.1 Formule de Marc Joye pour F

Considérons :

$$y = \lambda(x - x_1) + y_1$$

et :

$$dy^2 = x^3 + ax + b$$

Alors :

$$d(\lambda(x - x_1) + y_1)^2 = x^3 + ax + b$$

En développant :

$$x^3 - d\lambda^2 x^2 + (a + 2d\lambda^2 x_1 - 2d\lambda y_1)x + (b - d\lambda^2 x_1^2 + 2d\lambda y_1 x_1 - dy_1^2) = 0$$

Si x_1 , x_2 et x_3 les racines du polynômes, les relations de Viète donnent :

$$\boxed{x_1 + x_2 + x_3 = d\lambda^2}$$

Ainsi :

$$\begin{aligned} x_1 + x_2 + x_3 &= d \frac{(y - y_1)^2}{(x - x_1)^2} \\ x_1 + x_2 + x_3 &= d \frac{(y^2 + y_1^2 - 2y_3 y_1)^2}{(x_3 - x_1)^2} \\ x_2 &= \frac{(dy^2 + dy_1^2 - 2dy_3 y_1)}{(x_3 - x_1)^2} - x_1 - x_3 \\ x_2 &= \frac{d(x^3 + ax^2 + b + x_1^3 + ax_1^2 + b - 2y_3 y_1)}{(x_3 - x_1)^2} - x_1 - x_3 \\ x_2 &= \frac{(x^3 + ax^2 + b + x_1^3 + ax_1^2 + b - 2dy_3 y_1) - x_1(x_3 - x_1)^2 - x_3(x_3 - x_1)^2}{(x_3 - x_1)^2} \\ x_2 &= \frac{-2dy_3 y_1 + 2b + (a + x_3 x_1)(x_3 + x_1)}{(x_1 - x_3)^2} \end{aligned}$$

d'où :

$$\boxed{y_1 = \frac{2b + (a + x_3 x_1)(x_3 x_1) - x_2(x_3 - x_1)^2}{2dy_3}}$$

alors que *lobster256* calcule :

$$y_1 = \frac{2b + (a + x_3 x_1)(x_3 x_1) - x_2(x_3 - x_1)^2}{2y_3}$$

Ainsi à la fin du calcul on obtient :

$$(x_X, y_X) = XZ_EXP(s, x_{pub}, y_{pub}, a, b, p)$$

où :

$$\begin{aligned} x_X &= x_T \\ y_X &= d \times y_T \end{aligned}$$

avec :

$$T = s \times Pub$$

5.2 Résoudre $T = s \times Pub$

Nous pouvons modifier la valeur de x_{pub} afin que l'ECDLP ne soit plus à résoudre sur la courbe E choisie par les auteurs mais sur une courbe qui nous sera beaucoup plus favorable (ordre friable), et sur laquelle les algorithmes de résolution seront beaucoup plus rapides.

On choisit une valeur de x_{pub} de sorte que le point ne se situe plus sur la courbe mais sur son *twist quadratique* (ainsi on conserve les paramètres a et b). Par exemple on peut remplacer le dernier octet de x_{pub} par 0.

Sur cette nouvelle courbe, l'ordre se factorise en des facteurs de tailles beaucoup plus raisonnables pour lesquels il est possible d'utiliser Pollard rho pour calculer s à partir de T et Pub .

5.2.1 Forme de Weierstrass

Pour manipuler le *twist quadratique* dans Sage, il est possible de le mettre sous la forme de Weierstrass.

Ainsi :

$$dy^2 = x^3 + ax + b$$

est transformée en :

$$Y^2 = X^3 + AX + B$$

avec :

$$A = a * d^2$$

$$B = b * d^3$$

et :

$$X = x * d$$

$$Y = y * d^2$$

6 Résoudre : $W = S + X$

Il s'agit de la somme de deux points :

```
# addition de points P + Q sur y^2 = x^3 + a x + b (mod p)
def point_add(P, Q, a, p):
    if P is INFINITY:
        return Q
    if Q is INFINITY:
        return P
    x1, y1 = P
```

```

x2, y2 = Q

if (x1 - x2) % p == 0:
    # soit P = Q, soit P = -Q -> infini
    if (y1 + y2) % p == 0:
        return INFINITY # P + (-P) = INFINITY
    # sinon ce sera le cas P == Q et y1 != -y2
if x1 == x2 and y1 == y2:
    # doublage
    if y1 % p == 0:
        return INFINITY
    num = (3 * x1 * x1 + a) % p
    den = (2 * y1) % p
    lam = (num * inv_mod(den, p)) % p
else:
    # addition de deux points distincts
    if (x2 - x1) % p == 0:
        return INFINITY
    num = (y2 - y1) % p
    den = (x2 - x1) % p
    lam = (num * inv_mod(den, p)) % p

x3 = (lam * lam - x1 - x2) % p
y3 = (lam * (x1 - x3) - y1) % p
return (x3, y3)

```

Le calcul ne prend pas en compte le fait qu'on manipule des points (pas de vérification d'appartenance à la courbe). On peut s'inspirer de l'article *Fault Attacks on ECC Signature Verification*.

W et S sont fixés. Nous cherchons un point X vérifiant :

$$\lambda = \frac{y_X - y_S}{x_X - x_S}$$

$$\lambda^2 = x_W + x_S + x_X$$

En développant :

$$\left(\frac{y_X - y_S}{x_X - x_S}\right)^2 = x_W + x_S + x_X$$

$$(y_X - y_S)^2 = (x_W + x_S + x_X)(x_X - x_S)^2$$

$$y_X^2 + y_S^2 - 2y_X y_S = (x_W + x_S + x_X)(x_X - x_S)^2$$

Comme nous l'avons vu précédemment, le point que l'on recherche est :

$$x_X = x_T$$

$$y_X = d \times y_T$$

d'où :

$$d^2 y_T^2 + y_S^2 - 2y_X y_S = (x_W + x_S + x_T)(x_T - x_S)^2$$

Le point T vérifie :

$$dy_T^2 = x_T^3 + ax + b$$

alors :

$$\begin{aligned}
d(x_T^3 + ax_T + b) + y_S^2 - 2y_X y_S &= (x_W + x_S + x_T)x_T - x_S^2 \\
-2y_X y_S &= (x_W + x_S + x_T)x_T - x_S^2 - d(x_T^3 + ax_T + b) - y_S^2 \\
4y_X^2 y_S^2 &= ((x_W + x_S + x_T)x_T - x_S^2 - d(x_T^3 + ax_T + b) - y_S^2)^2 \\
4d^2 y_R^2 y_S^2 &= ((x_W + x_S + x_T)x_T - x_S^2 - d(x_T^3 + ax_T + b) - y_S^2)^2
\end{aligned}$$

d'où :

$$0 = ((x_W + x_S + x_T)x_T - x_S^2 - d(x_T^3 + ax_T + b) - y_S^2)^2 - 4dy_S^2(x_T^3 + ax_T + b)$$

Sage se fera une joie de trouver les valeurs de x_T qui conviennent.

Dans certains cas, aucune solution n'existe. Il suffira alors de choisir un autre point x_W et recommencer.

7 Forger une signature

Nous avons maintenant toutes les briques nécessaires :

- m : message pour lequel on veut une signature
- n : ordre de la courbe elliptique,
- On fixe un x_W ,
- Calcul de r : $r = \text{hash}(x_W)$,
- Calcul de h : $h = \text{hash}(x_{\text{pub}} + y_{\text{pub}} + m)$,
- Calcul de e : $e = (r \oplus h) \bmod n$,
- Calcul de S : $S = e \times G$,
- Calcul de d coefficient pour le *twist quadratique*,
- Définition de la courbe F ,
- Recherche des racines du polynôme à partir de x_W et $S : T$ (si pas de solution on recommence avec un autre x_W),
- Transposition de T et du point de clef public sur F ,
- Résolution de l'ECDL $T_{\text{twisted}} = s \times \text{Pub}_{\text{twisted}}$: on obtient s .

Nous obtenons ainsi une signature (r, s) valide pour notre message.

8 Flag et suite...

Les fichiers *serialize.py* et *diode_dest.py* font référence à un type de message *UTILS_GET_FLAG_STEP3*. On crée alors un fichier comme précédemment, la signature en plus. On obtient ainsi le flag.

On en profite aussi pour créer un message de type *UPDATE_SIG_KEY* afin de mettre à jour la clef publique du serveur avec notre propre clef. On peut ainsi calculer des signatures pour des nouveaux messages sans problème.

Nous sommes maintenant capable de créer des fichiers signés dont le contenu est envoyé à *weapon_authent*.

