**Problem**

A cryptographic issue in OpenPGP.js <=4.2.0 allows an attacker who is able provide forged messages and gain feedback about whether decryption of these messages succeeded to conduct an invalid curve attack in order to gain the victim's ECDH private key.

Invalid Curve Attack (CVE-2019-9155)

* The implementation of the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm does not verify that the communication partner's public key is valid (i.e., that the point lies on the elliptic curve). This causes the application to implicitly calculate the resulting secret key not based on the specified elliptic curve but rather an altered curve. By carefully choosing multiple altered curves (and therefore the resulting public key), and observing whether decryption fails, an attacker can extract the victim's private key.
* This attack requires the attacker to be able to provide multiple manipulated messages and to observe whether decryption fails.

Proof of concept : <https://packetstormsecurity.com/files/154191/OpenPGP.js-4.2.0-Signature-Bypass-Invalid-Curve-Attack.html>

The script in invalid\_curve\_attack.js demonstrates this issue. Note that since OpenPGP uses only the x-coordinate of the secret point, the oracle succeeds both when it calculates the same point as the attacker or when it calculates its inverse point. Therefore, the Chinese Remainder Theorem cannot be directly applied to the result. Instead, the script demonstrates that the remainder of the private key matches the corresponding gathered remainder. In order to fully implement this attack, an attacker could verify the results using other public key points with a non-prime order.

<https://github.com/openpgpjs/openpgpjs/pull/853>

<https://github.com/openpgpjs/openpgpjs/pull/853/commits/7ba4f8c655e7fd7706e8d7334e44b40fdf56c43e>

**ecdh.js**

async function kdf(hash\_algo, X, length, param) { **Il manque la curve**

const digest = await hash.digest(hash\_algo, util.concatUint8Array([

new Uint8Array([0, 0, 0, 1]),

new Uint8Array(X),

param

]));

return digest.subarray(0, length);

}

async function encrypt(oid, cipher\_algo, hash\_algo, m, Q, fingerprint) {

const { V, Z } = await genPublicEphemeralKey(oid, cipher\_algo, hash\_algo, Q, fingerprint);

return {

V: new BN(V),

C: aes\_kw.wrap(Z, m.toString())

}

/\*\*

\* Generate ECDHE ephemeral key and secret from public key

\*

\* @param {module:type/oid} oid Elliptic curve object identifier

\* @param {module:enums.symmetric} cipher\_algo Symmetric cipher to use

\* @param {module:enums.hash} hash\_algo Hash algorithm to use

\* @param {Uint8Array} Q Recipient public key

\* @param {String} fingerprint Recipient fingerprint

\* @returns {Promise<{V: Uint8Array, Z: Uint8Array}>} Returns public part of ephemeral key and generated ephemeral secret

\* @async

\*/

async function genPublicEphemeralKey(oid, cipher\_algo, hash\_algo, Q, fingerprint) {

const **curve** = new Curve(oid);

const param = buildEcdhParam(enums.publicKey.ecdh, oid, cipher\_algo, hash\_algo, fingerprint);

cipher\_algo = enums.read(enums.symmetric, cipher\_algo);

const v = await curve.genKeyPair();

Q = curve.keyFromPublic(Q);

const S = v.derive(Q);

const V = new Uint8Array(v.getPublic());

const Z = await **kdf**(hash\_algo, S, cipher[cipher\_algo].keySize, param); **//Curve pas dérivé**

return { V, Z };

}

async function genPrivateEphemeralKey(oid, cipher\_algo, hash\_algo, V, d, fingerprint) {

const **curve** = new Curve(oid);

const param = buildEcdhParam(enums.publicKey.ecdh, oid, cipher\_algo, hash\_algo, fingerprint);

cipher\_algo = enums.read(enums.symmetric, cipher\_algo);

V = **curve**.keyFromPublic(V);

d = **curve**.keyFromPrivate(d);

const **S** = d.derive(V);

return kdf(hash\_algo, **S**, cipher[cipher\_algo].keySize, param); **// Il manque la curve. (aucun cross-validation)**

}

//Ici comme on peut le constater, on ne décrypte pas en utilisant la **curve** de la clé publique.

async function decrypt(oid, cipher\_algo, hash\_algo, V, C, d, fingerprint) {

const Z = await genPrivateEphemeralKey(oid, cipher\_algo, hash\_algo, V, d, fingerprint);

return new BN(aes\_kw.unwrap(Z, C));

}

**Fix**

**ecdh.js**

async function kdf(hash\_algo, S, length, param, **curve**, **compat**) {

**const len = compat** ?

S.byteLength() :

**curve.curve.curve**.p.byteLength();

// Note: this is not ideal, but the RFC's are unclear

// https://tools.ietf.org/html/draft-ietf-openpgp-rfc4880bis-02#appendix-B

const X = **curve.curve.curve**.type === 'mont' ?

S.toArrayLike(Uint8Array, 'le', len) :

S.toArrayLike(Uint8Array, 'be', len);

const digest = await hash.digest(hash\_algo, util.concatUint8Array([

new Uint8Array([0, 0, 0, 1]),

X,

param

]));

return digest.subarray(0, length);

}

async function encrypt(oid, cipher\_algo, hash\_algo, m, Q, fingerprint) {

const curve = new Curve(oid);

const { V, S } = await genPublicEphemeralKey(curve, Q);

**const param = buildEcdhParam(enums.publicKey.ecdh, oid, cipher\_algo, hash\_algo, fingerprint);**

**cipher\_algo = enums.read(enums.symmetric, cipher\_algo);**

const Z = await **kdf**(hash\_algo, S, cipher[cipher\_algo].keySize, param, **curve, false**);

const C = aes\_kw.wrap(Z, m.toString());

return { V, C };

}

/\*\*

\* Generate ECDHE ephemeral key and secret from public key

\* @param {Curve} curve Elliptic curve object

\* @param {Uint8Array} Q Recipient public key

\* @returns {Promise<{V: Uint8Array, S: BN}>} Returns public part of ephemeral key and generated ephemeral secret

\* @async

\*/

async function genPublicEphemeralKey(curve, Q) { //Logique transféré dans **encrypt()**

const v = await curve.genKeyPair();

Q = curve.keyFromPublic(Q);

const V = new Uint8Array(v.getPublic());

const S = v.derive(Q);

return { V, S };

}

async function genPrivateEphemeralKey(curve, V, d) { //Logique transféré dans **decrypt()**

V = curve.keyFromPublic(V);

d = curve.keyFromPrivate(d);

return d.derive(V);

}

async function decrypt(oid, cipher\_algo, hash\_algo, V, C, d, fingerprint) {

const curve = new Curve(oid);

const S = await genPrivateEphemeralKey(curve, V, d);

const param = buildEcdhParam(enums.publicKey.ecdh, oid, cipher\_algo, hash\_algo, fingerprint);

cipher\_algo = enums.read(enums.symmetric, cipher\_algo);

try {

const Z = await kdf(hash\_algo, S, cipher[cipher\_algo].keySize, param, **curve**, false);

return new BN(aes\_kw.unwrap(Z, C));

} catch(e) {}

// Work around old OpenPGP.js bug.

const Z = await kdf(hash\_algo, S, cipher[cipher\_algo].keySize, param, **curve**, true);

return new BN(aes\_kw.unwrap(Z, C));

}