Crypto Layered signatures

5 AOUT

NOM DE LA SOCIÉTÉ

Créé par : JOL

Cryptography

Layered signatures

Comment ça marche :

Voici l'explication pas à pas de ce que fait le script (et, plus largement, de l'épreuve), puis pourquoi l'attaque fonctionne et comment elle résout le challenge à tous les coups.

1) Ce que met en place le challenge

1. Paramètres du groupe.
   Le script fixe un grand premier sûr p (≃ 1024 bits). "Premier sûr" signifie $p = 2 \cdot q_s + 1$ avec q_s premier ; le groupe multiplicatif modulo p est alors cyclique d'ordre $p-1$. C'est le cadre usuel des schémas à logarithme discret (Diffie–Hellman/Schnorr).
   https://en.wikipedia.org/wiki/Safe_and_Sophie_Germain_primes
2. Clés Schnorr « classiques ».
   On choisit un générateur g = 25 et une clé privée aléatoire s. La clé publique est gs = g^s mod p. C'est le cadre standard de Schnorr (hash e = H(r‖m), signature (s,e) où $s = k + x \cdot e$). **https://en.wikipedia.org/wiki/Schnorr_signature**
3. Un petit module indépendant q.
   Le serveur génère aussi un autre premier q (≈ 100 bits) qui n'a rien à voir avec p/q_s et ne sert qu'au contrôle final (m % q).
4. "Sous-signature Schnorr" correcte.
   La fonction weird_schnorr_sign(m, x) est une implémentation fidèle de Schnorr :

- k aléatoire, r = g^k,
- e = H(r‖m),
- $S = (k + x \cdot e) \bmod ((p-1)/2)$.
  La vérification refait r' = g^S·y^{−e} et compare le hash H(r'‖m).

5. Un emballage "maison" dangereux.
   La fonction sign(m, s) fabrique un nombre a très spécial :

   $a = (p-1-s) \cdot p - (q \cdot r + sta) \cdot (p-1)$

   puis signe a avec la sous-signature Schnorr ci-dessus. Elle prépare aussi 4 petits entiers t_i et une chaîne de hachages cc[i] (à base de SHA256) qui servent à un calcul mp. L'invariant voulu est que le vérificateur obtienne au final mp % q == m % q (donc « le même reste modulo q »).

6. Ce que donne le serveur à l'attaquant.
   À chaque connexion, il affiche :

- q, gs,
- une signature-démo sm (qui contient notamment a),
- leak = (flag % q),
  puis il demande « une signature valide du flag ».

2) Pourquoi la construction fuit la clé privée

Regardez a mod (p−1). Comme p ≡ 1 (mod p−1) :

a = (p−1−s)·p − (…)·(p−1)

  ≡ (p−1−s)·1 − 0

  ≡ −s   (mod p−1).

Donc a ≡ −s (mod p−1). Or a est révélé en clair dans la signature-démo sm. Une équation, une inconnue → on retrouve immédiatement la clé privée :

ini

s = (−a) mod (p−1).

L'attaque ne casse pas Schnorr : elle exploite une mauvaise "couche" par-dessus un schéma sain. (De même, dans Schnorr/EdDSA « pur », la moindre fuite/reutilisation de nonce k compromet la clé ; ici on fuit carrément une relation linéaire avec s.)
https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8214B.ipd.pdf

3) Pourquoi on peut signer le flag sans le connaître

La fonction verify(m, sig, gs) fait deux choses :

- (i) elle vérifie la sous-signature Schnorr sur a (avec gs) — ça passe, puisqu'on a signé a nous-mêmes avec la clé récupérée.
- (ii) elle reconstruit mp avec a, les t_i et la chaîne de hachages cc. Par construction, on a mp % q == m % q.

Le point crucial : la vérification ne regarde que m % q.
Or le serveur nous donne déjà leak = flag % q. Il suffit donc de signer leak (au lieu du vrai flag) : la vérification le considérera équivalent au flag modulo q et acceptera. On n'a jamais eu besoin de connaître le flag ; c'est le serveur qui l'affiche une fois la signature validée.

4) Ce que fait le solveur (script) pas à pas

1. Connexion au service, lecture des lignes jusqu'au prompt (les textes pouvant varier).
2. Extraction de q, gs, sm, leak.
3. Clé privée : récupérer a = sm[2] puis calculer s = (−a) mod (p−1) (propriété du groupe multiplicatif modulo p, d'ordre p−1, donc les exposants se prennent modulo p−1).
https://crypto.stanford.edu/pbc/notes/numbertheory/gen.html
4. Sous-signature fidèle : reproduire exactement la définition de Schnorr :
k aléatoire, r=g^k, e = H(r‖m) (concaténation d'octets), S = (k + s·e) mod ((p−1)/2).
5. Signature "complète" : reconstituer sign(m, s) du challenge (recréer la chaîne cc, sta, puis recalculer a comme dans le code, et enfin emballer [S, e, a, t0, t1, t2, t3]).
6. Cible à signer : prendre m = leak (car leak ≡ flag (mod q)).
7. Envoi de la signature en CSV au service.
8. Lecture : le service vérifie, trouve que c'est bon pour le flag, et affiche le flag en clair.

5) Pourquoi tout cela est correct (rappels & sources)

- Schnorr standard : e = H(r‖M), S = k + x·e; vérif. par r' = g^S·y^{−e} et recomputation du hash. C'est le mécanisme copié dans weird_schnorr_sign.
- Groupe modulo p : pour un premier p, le groupe multiplicatif ℤ_p^* est cyclique d'ordre p−1. Si g^{a+s} ≡ 1 (mod p), alors a+s ≡ 0 (mod p−1) (exposants pris modulo l'ordre). C'est l'algebra derrière a ≡ −s (mod p−1). https://www.di-mgt.com.au/multiplicative-group-mod-p.html
- Premiers sûrs : on choisit souvent p = 2·q_s + 1 pour assurer un grand sous-groupe d'ordre premier ; ici c'est bien le cas du p fourni. Wikipédia
- "Ne bricolez pas un schéma sûr" : en Schnorr/EdDSA, la moindre fuite structurelle (ex. nonce biaisé/réutilisé) mène à une récupération de clé ; ici c'est pire : on publie une quantité linéairement liée à la clé (a ≡ −s).

---

6) En une phrase

Le challenge offre par erreur une équation linéaire sur la clé privée (a ≡ −s (mod p−1)), ce qui permet de reconstruire s, puis de signer leak = flag % q (la vérif. ne regarde que m % q) ; la signature est acceptée pour le flag, que le serveur affiche aussitôt. WikipédiaWikipédiadi-mgt.com.auPublications NIST

Références utiles

- Définition formelle de Schnorr (hash H(r‖M), calcul de (S,e) et vérification).
- Premiers sûrs (définition et usage en DL).
- Groupe multiplicatif modulo p (cyclicité, ordre p−1, exposants modulo p−1). **di-mgt.com.au**
- NIST IR 8214B : danger des fuites/biais de nonce dans Schnorr/EdDSA (rappel « ne modifiez pas les schémas »). **https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8214B.ipd.pdf**

Principes ou mécanismes

1. **Lecture du banner** : on boucle tant qu'on n'a pas vu un : dans la ligne (le prompt "signature for flag:" peut varier).
2. **Extraction** (regex) : on récupère les nombres entiers quels que soient les espaces ou retours ligne.
3. **Clé secrète** : s dérive directement de a = sm[2], car le défi fuit a ≡ −s (mod p-1).
4. **Fonctions identiques** : weird_schnorr_sign et sign sont recopiées bit-à-bit, en particulier la concaténation d'octets long_to_bytes(r) + long_to_bytes(m) pour calculer e ; c'est indispensable pour que verify() calcule exactement la même valeur.
5. **Signature envoyée** : elle est acceptée tant que le serveur n'a pas régénéré de nouveaux paramètres (nouvelle connexion ⇒ nouveaux nombres ⇒ il faut relancer le solveur).

La version ci-dessous reproduit bit-à-bit le code original ; elle fonctionne à chaque nouvelle connexion car elle :

- lit les paramètres (q, gs, sm, leak) quels que soient les textes du prompt ;
- reconstitue la clé secrète s = (-sm[2]) mod (p-1) ;
- appelle une ré-écriture fidèle des deux fonctions du défi : weird_schnorr_sign et sign ;
- envoie immédiatement la signature fraîche.

Points de contrôle

Vérification	Pourquoi ?
pwntools installé (pip install pwntools)	simplifie le recvline() et la gestion TCP docs.pwntools.com
Aucun « … » ni caractère parasite dans les entiers	Python lèverait ValueError ou la signature serait fausse.
Le solveur est relancé à chaque **nouvelle** connexion	q, gs, sm, leak changent à chaque run côté serveur.
SHA-256 appliqué sur **r‖m** et non r+m	concaténation d'octets, sinon le hash diverge Stack Overflow
randbits(1024) pour le nonce k	comparable à getRandomInteger(1024) du défi.

En relançant maintenant :

bash
python3 solve_layered_sig.py

Il est censé apparaître le flag officiel renvoyé par le serveur — et pas de « Verification failed! ».

The flag
wwf{d1d_y0u_kn0w_tH3_pow3r_0f_W@gner?4b84gb5v783439u392093hr2b293rv3f283ru2bf3}