Pablo Rauzy
pablo.rauzy@univ-paris8.fr
https://pablo.rauzy.name/teaching/sese

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Sécurité et systèmes embarqués

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / M1 informatique

Séance 8 (TP) :

N'oubliez pas :
- Les TPs doivent être rendus par courriel au plus tard le lendemain du jour où ils ont lieu avec "[sese]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[sese] TP8 Rauzy".
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar gzippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple "rauzy-tp8").
- Si l'archive est lourde (> 1 Mo), merci d'utiliser `https://bigfiles.univ-paris8.fr/`.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net!**

Dans ce TP :
- CRT-RSA.
- Attaque par injection de faute.

Exercice 0.

Récupérations des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/sese/s8-tp/`.
2. Récupérez les fichiers nécessaires depuis la page web du cours, ou directement en ligne de commande avec `wget https://pablo.rauzy.name/teaching/sese/seance-8_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l'archive (par exemple avec la commande `tar xzf seance-8_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv sese_seance-8_files votre-nom`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

RSA.

Cet exercice est à faire en Python.

1. On a revu RSA dans le cours de la séance 7. N'hésitez pas à aller y jeter un œil pour vous rafraîchir la mémoire.
   On va commencer par générer une paire de clef publique/privée pour RSA.
   → Générez deux nombres premiers de 1024 bits que vous enregistrerez dans deux variables p et q.
   Vous pouvez pour cela vous servir par exemple de la commande `openssl prime` (à l'aide de la documentation disponible sur le web).
2. → Calculez $N = p \cdot q$.
3. On veut maintenant calculer les exposants d et e de tel sorte à ce que (N, e) soit la clef publique, et (N, d) soit la clef privée.
   On utilise souvent $e = 65537$.
   → Calculez d pour ce e et votre N (on rappelle que d et e sont inverses l'un de l'autre modulo $\varphi(N)$).
   À nouveau n'hésitez pas à chercher de l'aide sur le web...
4. → Assignez à une variable m une valeur aléatoire (avec le module `random` de la bibliothèque standard de Python) de 2048 bits maximum.
5. On veut maintenant faire un calcul de signature RSA.
   → Implémentez la fonction `modexp(b, e, m)` $= b^e \bmod m$ en Python, en utilisant bien sûr l'algorithme rapide "square-and-multiply" vu en cours.
6. Utilisez votre code pour calculer la signature de votre message m.
7. → À l'aide de la clef publique, vérifiez que votre signature est correct.

Exercice 2.

Optimisation de RSA avec le CRT.

1. → En utilisant la fonction `time` du module `time` de Python, mesurez le temps que prend votre code Python pour faire 100 fois le calcul de la signature.
2. Toujours dans le cours de la séance 8, nous avons vu l'optimisation de RSA avec le théorème des restes chinois : CRT-RSA

   → La clef publique de change pas pour CRT-RSA. En revanche, quelles sont les valeurs nécessaires dans la clef privée ?
3. → Dans votre code Python, calculez les valeurs manquantes pour la clef privée de CRT-RSA.
4. → Calculez s_p, s_q, et s_{crt} la signature de votre m en utilisant CRT-RSA.
5. → À nouveau, vérifiez que votre signature est correcte, par exemple en la comparant avec celle déjà vérifiée qui a été calculée sans l'optimisation CRT.
6. → Mesurez maintenant le temps que prend votre code Python pour faire 100 fois le calcul de la signature avec l'optimisation CRT.

   Que constatez-vous ?

Exercice 3.

Attaque par injection de faute sur CRT-RSA.

1. Comme on l'a vu en cours, le soucis avec CRT-RSA est sa vulnérabilité aux attaques par injection de faute.

   Dans `crtrsa.asm`, il y a une implémentation de CRT-RSA utilisant l'exponentiation modulaire qu'on a programmé au TP de la séance 7.

   Voici la clef publique utilisée pour cette implémentation : $(N, e) = (47775493107113604137, 17)$.

   Vous pouvez lancer le calcul avec `python3 bench.py -i crtrsa.asm /dev/null` : l'option `-i` active la possibilité de faire des injections de fautes (avec `^C`, et on utilise `/dev/null` comme log de consommation car on ne s'y intéresse pas cette fois-ci.

   Lorsque vous faite `^C` pendant le calcul, vous aurez le choix du type de faute à injecter dans le calcul avant que celui-ci ne reprenne.

   → Essayez d'injecter une ou des fautes de différentes natures de manière à avoir un résultat faux à la fin du calcul.
2. → Factorisez N en p et q à l'aide de l'attaque BellCoRe comme on l'a vue en cours à la séance 7.
3. → Retrouvez la clef privée.