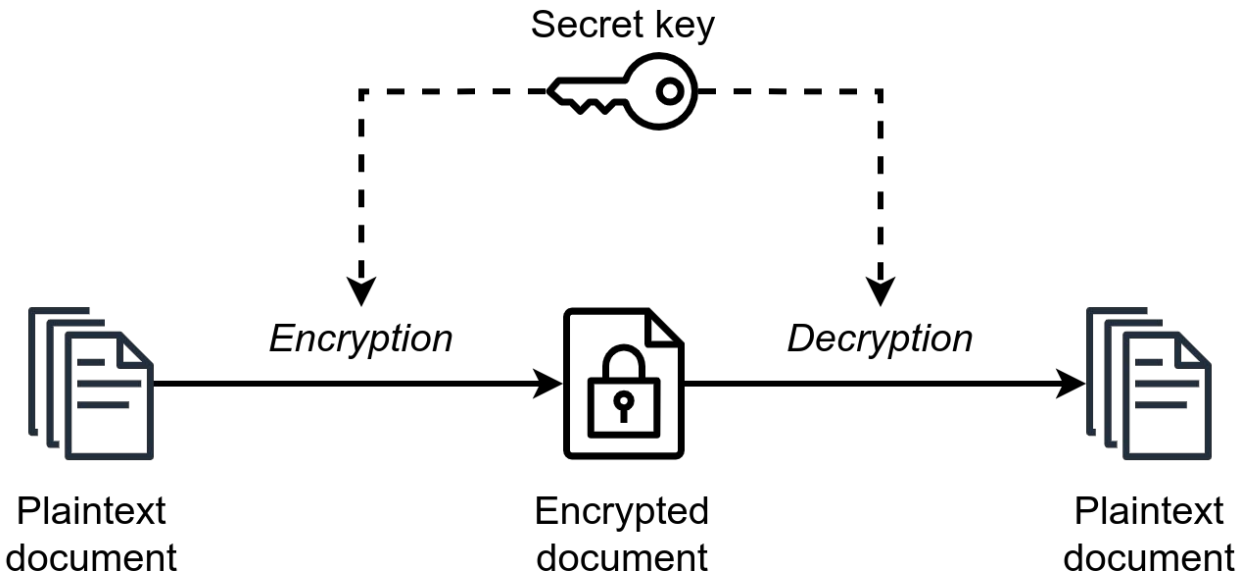
Μη Συμμετρική Κρυπτογραφία

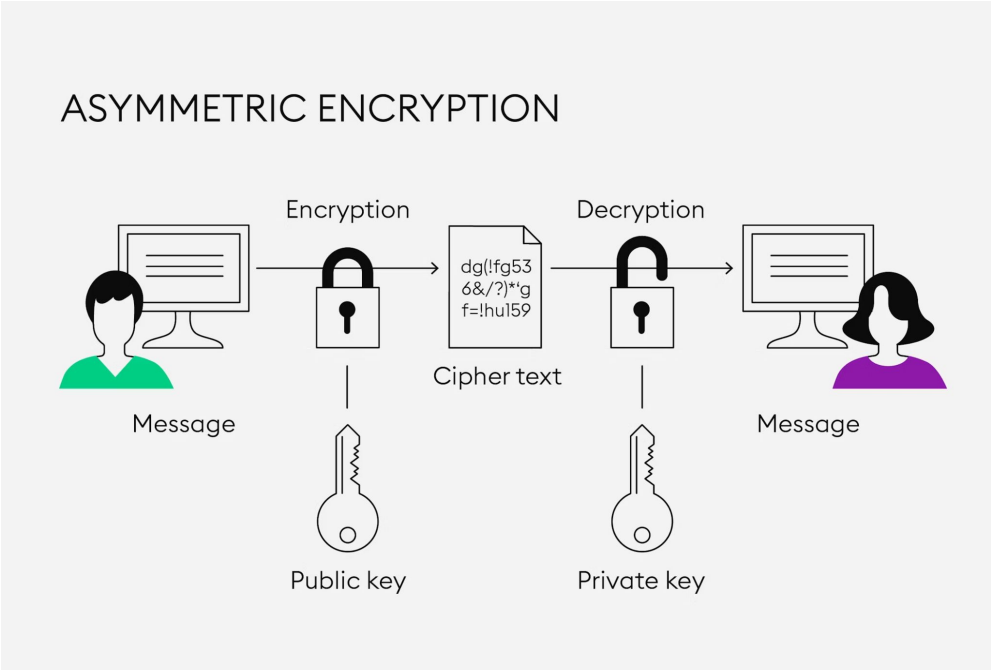
Συμμετρική κρυπτογραφία = κοινό κλειδί



Το πρόβλημα:

- Πως θα συμφωνήσουν η Αλίκη και ο Bob για το **ίδιο** ιδιωτικό κλειδί?
- Αν απλά το στείλουν **unencrypted** τότε κάποιος που κάνει eavesdrop θα το υποκλέψει και θα μπορεί στη συνέχεια να αποκρυπτογραφεί τα **encrypted** μηνύματα.
- Η λύση? **Μη-συμμετρική** κρυπτογραφία(aka κρυπτογραφία **δημοσίου κλειδιού**)
- Στη μη-συμμετρική κρυπτογραφία έχουμε **ζεύγος ιδιωτικού-δημοσίου** κλειδιού.
- **Encryption** με το δημόσιο κλειδί(**public**, γνωστό σε όλους)
- **Decryption** με το ιδιωτικό κλειδί(**private,** γνωστό **μόνο** στο “νόμιμο” παραλήπτη)
- Και τα δύο κλειδιά είναι του ίδιου ατόμου(του παραλήπτη)

ΜΗ Συμμετρική Κρυπτογραφία



ΜΗ Συμμετρική Κρυπτογραφία

- Είναι σημαντικά πιο αργή από τη συμμετρική.
- Για αυτό και χρησιμοποιείται για την ανταλλαγή του αρχικού, πρώτου, κοινού μυστικού με ασφαλή τρόπο, το οποίο στη συνέχεια θα χρησιμοποιηθεί ως το κλειδί για **συμμετρική** κρυπτογράφηση, η οποία είναι αποδοτικότερη.
- Έχει και αρκετές ακόμα χρήσεις με πιο σημαντική τις ψηφιακές υπογραφές(story for another time)
- Συνήθως, οι μη συμμετρικοί αλγόριθμοι βασίζονται σε κάποιο δύσκολο (μαθηματικό) πρόβλημα, μια “trapdoor” function που είναι εύκολο να υπολογιστεί προς τη μια κατεύθυνση αλλά δύσκολο προς την αντίθετη χωρίς τη γνώση της “καταπακτής”(trapdoor)
- Πολλά από αυτά τα “δύσκολα” μαθηματικά προβλήματα θα γίνουν εύκολα όταν φτιαχτούν κβαντικοί υπολογιστές :(

RSA(Rivest-Shamir-Adleman)

- Ο πιο γνωστός αλγόριθμος κρυπτογράφησης δημοσίου κλειδιού.
- Φτιάχτηκε από τους Rivest, Shamir, Adleman
- Χρησιμοποιείται πλέον κυρίως για ψηφιακές υπογραφές/certificates
- Βασίζεται στο δύσκολο μαθηματικό πρόβλημα της **παραγοντοποίησης** μεγάλων αριθμών

RSA(συγκεκριμένα)

- Ο Bob διαλέγει 2 τυχαίους **μεγάλους**(πιθανώς χιλιάδες bits) **πρώτους** αριθμούς **p, q**
- Υπολογίζει το **public modulus** **n = p*q**
- Διαλέγει ένα δημόσιο εκθέτη(**public exponent**) **e = 65537 ή 3**
- Το ζεύγος (n, e) αποτελεί το **δημόσιο κλειδί** του Bob.
- Ο Bob υπολογίζει τον ιδιωτικό εκθέτη(**private exponent) d** έτσι ώστε να ισχύει **e*d = 1 (mod φ(n))**
- **φ()** είναι η συνάρτηση του Euler(θα πούμε περισσότερα σε επόμενη διαφάνεια), μπορεί να την υπολογίσει για το n **ΜΟΝΟ** επειδή ξέρει τους πρώτους **p**, **q**!!!
- Το ζεύγος (n, d) αποτελεί το **ιδιωτικό κλειδί** του Bob.(ok technically το n είναι public information άλλα η “σύμβαση” είναι να μπαίνει και στο ιδιωτικό κλειδί)

Συνάρτηση του Euler

- Η τιμή φ(m) εκφράζει το πλήθος των αριθμών στο εύρος [1, m] που έχουν GCD με το m ίσο με 1
- Αν $m = p_1^{k_1} * p_2^{k_2} * \dots * p_i^{k_i}$ η ανάλυση του m σε (διακεκριμένους) πρώτους παράγοντες τότε $\varphi(m) = \varphi(p_1^{k_1}) * \varphi(p_2^{k_2}) * \dots * \varphi(p_i^{k_i})$
- Αν $m = p^k$ είναι δύναμη πρώτου αριθμού τότε $\varphi(m) = p^{(k-1)} * (p - 1)$
- Από τα 2 παραπάνω προκύπτει για το RSA ότι $\varphi(n) = \varphi(p)*\varphi(q) = (p - 1)*(q - 1)$
- Η παραγοντοποίηση μεγάλων αριθμών όπως το n είναι δύσκολη άρα **ΑΝ ΟΛΑ ΕΧΟΥΝ ΠΑΕΙ ΚΑΛΑ** μόνο ο Bob μπορεί να υπολογίσει το φ(n) το οποίο χρειάζεται για να βρει κανείς το d για το οποίο ισχύει $e*d = 1 \pmod{\varphi(n)}$
- (το πως βρίσκει το d γίνεται μέσω του επεκτεταμένου αλγόριθμου του ευκλείδη ή egcd/xgcd, αν θέλει κανείς μπορεί να το ψάξει αλλά στην python γίνεται απλά με pow(e, -1, phi))

RSA - Encryption/Decryption

- Αν κάποιος θέλει να στείλει κάτι στον Bob τότε πρώτα μετατρέπει το μήνυμα του σε αριθμό(στην python με bytes_to_long()), έστω **m** ο αριθμός που αντιστοιχεί στο μήνυμα.
- Υπολογίζει το ciphertext **c** ως **c = m^e (mod n)** (σε python c = pow(m, e, n)) και το στέλνει στο Bob.
- Ο Bob υπολογίζει το αρχικό **m** κάνοντας **m = c^d (mod n)** (σε python m = pow(c, d, n)).
- Το μετατρέπει απο αριθμό σε bytes.

- Απαραίτητη προϋπόθεση είναι m < n

RSA- Απόδειξη Ορθότητας(feel free to skip)

- Για να πεπειστούμε ότι δουλεύει αρκεί να δείξουμε ότι c^d είναι όντως ίσως με m(για συντομία δε γράφουμε το “(mod n)” στις πράξεις αλλά υποθέστε ότι είναι εκεί)
- c = m^e => c^d = (m^e)^d = m^(e*d) (σχέση 1)
- Το θεώρημα του Euler μας λέει ότι a^(φ(n)) = 1 (mod n)
- Επίσης, έχουμε e*d = 1 (mod φ(n)), αν “βγάλουμε” το mod αυτό γράφεται ως e*d = 1 + k*φ(n), για κάποιο ακέραιο k.
- Αντικαθιστούμε στη σχέση 1 και προκύπτει c^d = m^(1 + k*φ(n)) = m * (m^φ(n))^k =>(λόγω Euler προκύπτει:) c^d = m * 1^k = m, αποδείχτηκε!
- (Σημείωση για όσους θέλουν να ψάξουν κάτι παραπάνω: φ(n) είναι η **τάξη** της **ομάδας**)

RSA - Ασφάλεια/Ασφάλεια στα ctf

Υπάρχουν (πολύ χονδρικά) 4 τρόποι για να επιτεθεί κανείς στο RSA:

- Παραγοντοποίηση n -> υπολογισμός φ(n) -> υπολογισμός d -> decrypt message
- (κατευθείαν) υπολογισμός φ(n) -> υπολογισμός d -> decrypt message
- (κατευθείαν) υπολογισμός d -> decrypt message
- (κατευθείαν) decrypt message, αυτό είναι αρκετά διαφορετικού τύπου από τα άλλα 3 και απαιτεί κάποιες προϋποθέσεις

Γενικά, οι πρώτοι 3 τρόποι συνοψίζονται ως **επίθεση στο δημόσιο κλειδί**(και έπειτα αφού έχεις το d μπορείς να κάνεις decrypt **οποιοδήποτε** μήνυμα) ενώ ο τελευταίος είναι **επίθεση στο μήνυμα**, και μπορεί να γίνεται για λίγα/μόνο το συγκεκριμένο μήνυμα(αρκετές φορές η τιμή του δημόσιου εκθέτη e = 3 είναι ένδειξη ότι ο τελευταίος τρόπος θα είναι επιτυχής)

RSA - Basic small exponent attack

Έστω:

- n = 351686722502391049115805910033587067017529898124696998630958253188242602323463
- e = 3
- c = m^e % n = 125

Ποιο είναι το m? Προφανώς m = 5 = ∛125

- Αν τα e και m είναι μικρά τότε ίσως το m^e να μην έχει “περάσει” καμία φορά πάνω από το modulo n. Τότε μπορούμε να ανακτήσουμε το αρχικό m απλά παίρνοντας την e-οστή ρίζα του c(αδιαφορώντας για το n)

RSA - Λιγότερο basic small exponent attack

- Γενικά η σχέση c = m^e % n μπορεί να γραφτεί χωρίς το modulo ως:
    - **c = m^e - k*n => m^e = c + k*n**, για θετικό ακέραιο k
- Γνωστά είναι τα e, c, n και αγνωστα τα m, k(το m είναι το μήνυμα που μας ενδιαφέρει). Αν ξέραμε το k τότε μπορούμε να υπολογίσουμε την δεξιά μεριά της σχέσης και έπειτα πάλι να πάρουμε την **e-οστή ρίζα** της για να βρούμε το m.
- Συνηθώς η τιμή του k είναι ΤΕΡΑΣΤΙΑ(για μεγάλο m ή/και e), αν όμως είναι και τα 2 μικρά, τότε σε αντιστοιχία με την “basic” περίπτωση το m^e θα έχει “περάσει” λίγες φορες το n άρα το k θα είναι μικρό. Μπορούμε να **δοκιμάσουμε**, λοιπόν, διάφορες μικρές τιμές του k μέχρι να δουμε οτι η e-οστή ρίζα αυτού που υπολογίσαμε είναι μήνυμα που βγάζει νόημα(σα κάποιου είδους bruteforce).

RSA - Hastad’s Broadcast Attack

- Έστω 3 παραλήπτες B1, B2, B3 με RSA δημόσια κλειδιά (n1, e1), (n2, e2), (n3, e3) αντίστοιχα, για τα οποία ισχύει e1 = e2 = e3 = 3
- Έστω αποστολέας A που θέλει να στείλει **το ίδιο** μήνυμα m και στους 3 παραλήπτες. Για να μπορεί να το κάνει αυτό θα ισχύει σίγουρα m < n1, m < n2, m< n3
- Κάνοντας encrypt το m με τα 3 διαφορετικά κλειδιά θα προκύψουν:
    - c1 = m^3 % n1
    - c2 = m^3 % n2
    - c3 = m^3 % n3
- Αν τα c1, c2, c3 πέσουν στα χέρια κάποιου επιτιθέμενου τότε εκείνος μπορεί να ανακτήσει το αρχικό μήνυμα!!! Πώς? Η λύση είναι το Κινέζικο Θεώρημα Υπολοίπων(CRT)

CRT(Chinese Remainder Theorem)

- Έχοντας i σχέσεις:
  - $a_1 = X \% n_1$
  - $a_2 = X \% n_2$
  - …
  - $a_i = X \% n_i$
- Με γνωστά a_i και n_i, άγνωστο το X και n_1, n_2, …, n_i πρώτα **μεταξύ** τους τότε μπορεί να βρεθεί το A τέτοιο ώστε:
  - $A = X \% N$
- Όπου X ο άγνωστος από τις αρχικές σχέσεις και N το γινόμενο όλων των n_i.
- Για περισσότερες λεπτομέρειες και παραδείγματα δείτε τη wikipedia καθώς και τις διαφάνειες του crypto workshop στο NTUAH4CK 2.0(Γιάννης Κουμπιάς et al.)

RSA - Hastad’s Broadcast Attack(συνέχεια)

- Θυμίζουμε ότι έχουμε:
    - c1 = m^3 % n1
    - c2 = m^3 % n2
    - c3 = m^3 % n3
- Όμως αυτό θυμίζει πολύ τις σχέσεις του CRT όπου άγνωστος είναι το X = m^3. Εφαρμόζοντας το CRT θα πάρουμε μια τιμή M τέτοια ώστε M = m^3 % n1*n2*n3
- **Όμως**, m<n1, m<n2, m<n3. Πολλαπλασιάζοντας τις ανισότητες κατά μέλη προκύπτει m^3 < n1*n2*n3
- Άρα **m^3 % n1*n2*n3 = m^3!!!** Συνεπώς η τιμή M που μας έδωσε το CRT **είναι** το m^3, αυτούσιο! Όπως είπαμε όμως μερικές διαφάνειες πριν, σε αυτή την περίπτωση αρκεί να υπολογίσουμε την 3η ρίζα του M και θα πάρουμε το αρχικό μήνυμα m.

Fast exponentiation - Square and Multiply

- Στο RSA, για το encryption υψώνουμε ένα νούμερο στην e, συνήθως 65537, mod n. Στο decryption, υψώνουμε στο d το οποίο συνήθως είναι χιλιάδες bits(περίπου όσο το n).
- Αν για να υπολογίσουμε το m = c^d (mod n), αν κάναμε c*c*c*c*...*c θα έπαιρνε πάρα πολλή ώρα(δε θα τερμάτιζε ποτέ σε πολλές περιπτώσεις)
- Υπάρχει τρόπος όμως να υψώσεις σε δύναμη σε O(logn) χρόνο όπου n ο εκθέτης. Αυτός ο τρόπος είναι ο αλγόριθμος **square and multiply**.
- Έστω ότι θέλω να υπολογίσω το a_1^16. Μπορώ να υπολογίσω το a_2=a_1^2, έπειτα το a_3=a_2^2(=a_1^4), μετά το a_4=a_3^2(=a_1^8)και τέλος το επιθυμητό a_5=a_4^2(=a_1^16). Έτσι με μόνο 5 πολλαπλασιασμούς αντί για 15 μπορώ να υπολογίσω το αποτέλεσμα.
- Αν έχει mod απλά εφαρμόζουμε το mod σε κάθε ενδιάμεσο αποτέλεσμα

Fast exponentiation - Square and Multiply (2)

- Αν θέλω να υπολογίσω μια δύναμη που δεν είναι δύναμη του 2, πχ a_1^23 τότε το “σπάω” σε a_1^16 * a_1^4 * a_1^2 * a_1
- Βέβαια και με έτσι, αν (π.χ.)υπολογίσω πρώτα το a_1^16 και μετά το a_1^4 τότε πράξεις που χρειάστηκα για το a_1^4 τις είχα ήδη κάνει για τον υπολογισμό του a_1^16(βλ προηγούμενη διαφάνεια).
- Για να γλιτώσω αυτή την επανάληψη αχρείαστων πράξεων μπορώ να “αρχίσω” με τη μικρότερη δύναμη και να πάω προς τη μεγαλύτερη.

Square and multiply - Σε python

```python
def efficient_exp(base, exp, modulus= 0):
    q = base
    r = 1
    while exp > 0:
        if exp % 2 == 1:
            r *= q
            if modulus: r %= modulus
        q *= q
        if modulus: q %= modulus
        exp >>= 1
    return r
```

Το προαιρετικό τρίτο όρισμα είναι για να υπολογίσουμε τη δύναμη mod έναν αριθμό(όπως στο RSA) σε αντιστοιχία με την pow(base, exp, modulus) της python

Square and Multiply - Ο κίνδυνος των Side Channel

- Παρατηρείτε κάτι περίεργο?
- Η λούπα για τον υπολογισμό της δύναμης ουσιαστικά κάνει iterate τα bits του εκθέτη. Όταν το bit είναι 0 τότε κάνει μόνο **1 πράξη**(q = q**2 ή όπως είναι γραμμένο q *= q), ενώ όταν το bit είναι 1 κάνει **2 πράξεις**(r = r*q **και** q = q**2).
- Αν ένας επιτιθέμενος είχε πρόσβαση σε φαινομενικά άσχετες τιμές για τον υπολογιστή μας όπως η θερμότητα του επεξεργαστή, τα σήματα που εκπέμπει, κατανάλωση ενέργειας ή ακόμα και απλά το χρόνο που του πήρε για να εκτελέσει κάποιες λειτουργίες(όλα αυτά είναι τα λεγόμενα **side channels**) τότε **μπορεί να εξάγει πληροφορία για το εκάστοτε bit του εκθέτη**, αφού οι μία παραπάνω πράξη που εκτελεί όταν το το bit είναι 1 σε σχέση με 0 σημαίνει, περισσότερη θερμότητα, διαφορετικά σήματα, περισσότερη κατανάλωση ενέργειας, περισσότερος χρόνος εκτέλεσης κλπ.

Square and Multiply - Side Channel example

- Εισάγουμε μια τεχνητή καθυστέρηση στον προηγούμενο κώδικα για να γίνει αισθητή η διαφορά:

```python
import time

def efficient_exp(base, exp, modulus= 0):
    q = base
    r = 1

    while exp > 0:
        begintime = time.time()
        if exp % 2 == 1:
            r *= q
            if modulus: r %= modulus
            time.sleep(0.00001)
        q *= q
        if modulus: q %= modulus
        time.sleep(0.00001)
        print(time.time() - begintime, exp % 2)
        exp >>= 1
    return r
```

- Επίσης θεωρούμε ότι ο επιτιθέμενος μπορεί να λαμβάνει (με κάποιο τρόπο) μετρήσεις για το χρόνο που πήρε κάθε ένα iteration της λούπας
- Παίξτε με δικές σας τιμές με το διπλανό κώδικα(δίνεται στο άλλο αρχείο), θα δείτε την διαφορά στο χρόνο αναλόγως το bit του εκθέτη.
- Με αυτές τις μετρήσεις μπορεί ο επιτιθέμενος να ανακτήσει ένα-ένα όλα τα bits του εκθέτη οπότε αν ο εκθέτης είναι το **ιδιωτικό κλειδί d** του RSA τότε μπορεί να σπάσει το RSA!

A few resources

- Διαφάνειες crypto workshop NTUAH4CK 2.0
- RSA wikipedia (Τα λέει πολύ ωραια! Και έχει πληροφορίες για πολλά ακόμα attacks)
- CRT wikipedia (πάλι τα λέει ωραια)
- Cryptohack public key cryptography “course” (ωραία challenges για όποιον θέλει εξάσκηση)
- Το άλλο αρχείο που ανέβηκε έχει διάφορα, σχετικά με RSA παραδείγματα κώδικα σε python.