Security Project : RSA attacks

🚨 RSA Attack: When `n` is a Squared Prime 🔑

Attack Scenario 🎯

- **Assumption**: `n = p²` (where `p` is prime, a **fatal mistake** in RSA setup).
- **Goal**: Compute `φ(n)` to derive private key `d` and decrypt ciphertext.

🧠 Key Insight

- **Phi Formula**: `φ(n) = p² - p`
  Since
  `n = p²` → `p = √n` → `φ(n) = n - √n`.

Encryption script:

```
from Crypto.Util.number import inverse, long_to_bytes, bytes_to_long
import math

# RSA parameters
n = 53586080804400955002917713570816801620145134314731356537101445
e = 65537

# Calculate p and φ(n)
p = math.isqrt(n)
assert p * p == n, "n is not a perfect square!"
phi = n - p
```

```
#! New plaintext message
plaintext = b"square_root_factoring_is_not_safe"
m = bytes_to_long(plaintext)

# Compute new ciphertext (encryption)
ct = pow(m, e, n)

# For verification: decrypt the ciphertext
d = inverse(e, phi)
pt = long_to_bytes(pow(ct, d, n))
print(pt.decode())  # Expected output: square_root_factoring_is_not_safe

print("New ciphertext (ct):")
print(ct)
```

📜 Mathematical Solution in python:

```
phi = n - math.isqrt(n)  # p² - p
d = pow(e, -1, phi)      # Modular inverse of e mod φ(n)
pt = pow(ct, d, n)       # Decrypt using recovered private key
```

🐍 Python Snippet

```
from Crypto.Util.number import inverse, long_to_bytes
import math
n = 5358608080440095500291771357081680162014513431473135653710144590277434917394228854430847057207314097137755279937196825836691648738068420432884398280717899706947590808421622539552595905522830477287828129468451603348017820880681544530219367217102690509858050546920967387773217961533840248976155944934530681383412036737495140945460002536319029916171978475845196941521227 65
```

```
40698213352659492868523238193474215219586138022122437085812873
69759591768616510443703785390939901983362985729445127385708393
96588590096813217791191895941380464803377602779240663133834952
32931686239958195059058800637122133412821540919760323694259767
475672821223213405656271639915508010888110595276818919372882748
4667349378091100068224404684701674782399200373192433062767622
8412640554260353497690181172996205548039024904323396005664322
467958181674609161806473941691576472456035556927356308621487154
28791242764799469896924753470539857080767170052783918273180304
83531838817708967423164091033774378975097921620257322679424033
27978928682763094002539259322238955307141696481165690135816431
92341931800785254715083294526325980247219218364118877864892068
1859055874109771527379363107347122769566631921824876724746511032
40004173381041237906849437490609652395748868434296753449
e = 65537
ct=37350811549588685067866847777972094757964772391414050561504
82952057150429424161006021698778725930664535592359997554749811
6482361606076176572952763647842156895827853620388507541753360
4883375997033395709655793194053648567555581940342016367483977
45052066366643668759378360317324112271834599861540380793179348
582971486084112928698114664566192226761898874248183137322262057
85812469393176567333423366977023077559194141073280957397295065
9289062900723452692473846408981816989591200065299051503268705
62020349599245255581163938193521659385050962355187551095366815
5725481194126769773091274505285259240861571108401883139588714212
7248476949822073822332120068577119546720533106838970863054857
597919068861963823230140824359817121882266921086108463445661412
99792632863628183190801005255752954137866400313550318444614387
83992209183091648389631756475242924553389925446584511812619481
8584770052683869881533911408917296272296799525000346959956051
92763544013755480136325576328492426531522452475814381596272817
6783078497361191855362784863025598859546580076558754156324740
4170057325944424956666974479075428315568547593841231608458056
684081594408152905779328218012376228196123018227667723681169932
598539892322544769335915920319747350282315852863963251863786
phi = n - math.isqrt(n)
```

```
d = pow(e,-1,phi)
pt = long_to_bytes(pow (ct, d, n))
print ("The plain text is: ",pt.decode())
```

💡 Takeaway

- **Why It Works**: Poor RSA setup with `n = p²` leaks `φ(n)` → **total break**.
- **Real-World**: Always use **distinct primes** `p` and `q` for RSA. `n = p*q` → `φ(n) = (p-1)(q-1)` (secure if primes are hidden).

⚠️ **Lesson**: Never reuse primes or use `n = p²` in RSA!

🔐 Wiener Attack

📌 Overview

Wiener's attack is a cryptographic attack on **RSA encryption** that exploits **small private exponent (d)** values. It allows an attacker to recover the private key **d** when it is **too small relative to N** using **continued fractions**.

⚠️ When Does It Happen?

This attack occurs when the RSA private exponent **d** is **less than N^(1/4)**. Under such conditions, continued fraction expansion of **e/N** allows the attacker to efficiently approximate **d** using convergents.

🛠️ Encryption Code (Vulnerable RSA)

```
python
CopyEdit
#!/usr/bin/env python3
from Crypto.Util.number import getPrime, bytes_to_long
import math
```

```
FLAG = b"choose_d_wisely_next_time"
m = bytes_to_long(FLAG)

def get_huge_RSA():
    p = getPrime(1024)
    q = getPrime(1024)
    N = p * q
    phi = (p - 1) * (q - 1)
    # Select d as a 256-bit prime such that e = d^(-1) mod phi has bit-length eq
ual to N's bit-length.
    while True:
        d = getPrime(256)
        try:
            e = pow(d, -1, phi)
        except ValueError:
            continue
        if e.bit_length() == N.bit_length():
            break
    return N, e

N, e = get_huge_RSA()
c = pow(m, e, N)

print(f'N = {hex(N)}')
print(f'e = {hex(e)}')
print(f'c = {hex(c)}')
```

🎯 Attack Solution (Wiener's Attack)

```
python
CopyEdit
from Crypto.Util.number import long_to_bytes

# Here put N, c, and e
```

```
def wiener_attack(N, e):
    """Implementation of Wiener's Attack using continued fractions."""
    cf = continued_fraction(e / N)
    convergents = cf.convergents()

    for conv in convergents:
        k = conv.numerator()
        d = conv.denominator()

        if k == 0:
            continue

        # Check if (e * d - 1) is divisible by k
        if (e * d - 1) % k == 0:
            phi = (e * d - 1) // k
            b = N - phi + 1
            delta = b * b - 4 * N

            # Check if delta is a perfect square
            if delta >= 0 and is_square(delta):
                return d

    return None

# Run the attack
d = wiener_attack(N, e)

if d:
    print(f"Recovered d: {hex(d)}")

    # Decrypt
    m = power_mod(c, d, N)
    flag = long_to_bytes(m)

    print(f"Flag: {flag.decode()}")
```

```
else:
    print("Wiener's attack failed.")
```

✅ Key Takeaways

- 🔎 Wiener's attack works when **d is small** (i.e., d < N^(1/4)).
- 🔢 The attack uses **continued fractions** to approximate **d**.
- ⚡ If successful, it allows decryption without brute-forcing the private key.
- 🔥 **Best Mitigation:** Use a **larger d** to avoid vulnerability.

🔐 Low Public Exponent Attack

📌 Overview

The **Low Public Exponent Attack** is an RSA vulnerability that occurs when the public exponent **e** is **too small**, typically **e = 3** or **e = 65537**, and the plaintext is not properly padded. This allows an attacker to **recover the plaintext directly** using **nth-root techniques** or simple algebraic methods.

⚠️ When Does It Happen?

This attack occurs in two main scenarios:

1️⃣ **Unpadded Encryption:** If the message **m < N^(1/e)**, then simply computing the **e-th root** of **c = m^e mod N** directly reveals **m**.

2️⃣ **Broadcast Attack (Hastad's Attack):** If the **same message is sent to multiple recipients** with the same **e** but different **N**, an attacker can reconstruct the message using **Chinese Remainder Theorem (CRT)**.

🛠️ Encryption Code (Vulnerable RSA with Low e)

```
python
CopyEdit
#!/usr/bin/env python3
from Crypto.Util.number import getPrime, bytes_to_long
import math

FLAG = b"avoid_small_e_in_rsa"
m = bytes_to_long(FLAG)

def get_weak_RSA():
    p = getPrime(1024)
    q = getPrime(1024)
    N = p * q
    e = 3  # Low public exponent
    c = pow(m, e, N)
    return N, e, c

N, e, c = get_weak_RSA()

print(f'N = {hex(N)}')
print(f'e = {hex(e)}')
print(f'c = {hex(c)}')
```

🎯 Attack Solution (Cube Root Attack for e=3)

```
python
CopyEdit
from Crypto.Util.number import long_to_bytes
import gmpy2

# Here put N, c, and e

def low_exponent_attack(c, e):
```

```
    """If c = m^e and m < N^(1/e), we can recover m by computing e-th root."""
    m = gmpy2.iroot(c, e)[0]  # Compute integer cube root
    return long_to_bytes(int(m))

# Run the attack
message = low_exponent_attack(c, e)

print(f"Recovered plaintext: {message.decode()}")
```

✅ Key Takeaways

- 🔎 If **e is too small**, decryption can be done with simple math.
- 🏴 **No padding = High risk** → Always use **PKCS#1 or OAEP padding**.
- ⚡ If **m < N^(1/e)**, computing the **e-th root** directly reveals **m**.
- 🔥 **Best Mitigation:** Use **larger e** and implement **proper padding**.

🧑‍💻 Shor's Algorithm Attack Report

📌 Overview

Shor's Algorithm is a **quantum computing attack** that efficiently factors large numbers, breaking **RSA encryption** by computing the **private key (d)** from the **public key (N, e)**. While traditional computers take **exponential time**, **Shor's Algorithm** can solve this problem in **polynomial time** using a **Quantum Computer**.

⚠️ When Does It Happen?

1️⃣ **RSA security is based on the difficulty of factoring large numbers (N = p × q).**

2️⃣ **Classical computers take years to factorize N, but a quantum computer using Shor's Algorithm can do it in seconds.**

3️⃣ **This attack is currently theoretical** because large-scale quantum computers do not exist yet, but **future quantum advancements could render RSA insecure**.

🛠️ Encryption Code (RSA Setup for Attack)

```
python
CopyEdit
#!/usr/bin/env python3
from Crypto.Util.number import getPrime, bytes_to_long

FLAG = b"quantum_will_break_rsa"
m = bytes_to_long(FLAG)

def generate_RSA():
    p = getPrime(1024)
    q = getPrime(1024)
    N = p * q
    e = 65537  # Common public exponent
    c = pow(m, e, N)
    return N, e, c

N, e, c = generate_RSA()

print(f'N = {hex(N)}')
print(f'e = {hex(e)}')
print(f'c = {hex(c)}')
```

🎯 Attack Solution (Simulating Shor’s Algorithm)

```
python
CopyEdit
from sympy import factorint
from Crypto.Util.number import long_to_bytes
```

```
# Here put N and c

def shors_algorithm(N):
    """Simulating Shor's Algorithm by factoring N."""
    factors = factorint(N)  # This simulates a quantum factorization
    p, q = list(factors.keys())  # Extract prime factors
    return p, q

# Run the attack
p, q = shors_algorithm(N)
phi = (p - 1) * (q - 1)
d = pow(e, -1, phi)

# Decrypt
m = pow(c, d, N)
flag = long_to_bytes(m)

print(f"Recovered flag: {flag.decode()}")
```

✅ Key Takeaways

- 🧑‍💻 **Shor's Algorithm breaks RSA** by factoring **N = p × q** in **polynomial time** using **quantum computing**.
- ⚡ **Current RSA is safe**, but once **large-scale quantum computers** exist, it will be **completely broken**.
- 🔒 **Best Mitigation:** Use **Post-Quantum Cryptography** such as **Lattice-based** or **Elliptic Curve Cryptography (ECC)**.

🔮 Future of Cryptography?

With quantum computing advancing, the world is shifting toward **Post-Quantum Cryptography (PQC)** to secure data **against quantum threats**.

🧮 Mersenne Prime Factorization Attack

📌 Overview

In this attack, RSA encryption is broken by leveraging **Mersenne prime factorization** to decrypt a message. A **Mersenne prime** is a prime number of the form `2^n - 1`. If the modulus `n` in RSA encryption is composed of Mersenne primes, it can be factorized efficiently, compromising the security of RSA.

⚠️ When Does It Happen?

1️⃣ **RSA security is vulnerable when the modulus `n` is the product of Mersenne primes.**

2️⃣ **Mersenne primes can be factorized more easily** using certain algorithms, reducing the difficulty of breaking RSA encryption in these specific cases.

3️⃣ **This attack exploits the mathematical properties of Mersenne primes** to recover the private key and decrypt the ciphertext.

🛠️ Encryption Code (RSA Setup for Attack)

```
from Crypto.Util.number import bytes_to_long, long_to_bytes

# Given RSA parameters
n = 658416274830184544125027519921443515789888264156074733099244
0401262136824977140327981163992881765024628292557845259777229
0301871443430969810820838866476826275431642622065157662373161
7882923164117579624827261244506084274371250277849351631679441171
018418018498039996472549893150577189302871520311715179730714312
1814562450978484916697959972898306129880585239683848088228283
709001984892492433991651252192447537907797644662369651357935765
16193213175061401667388622228362042717054014679032953441034021
5068560170810626175723511954185058993887157097959920295590421197
```

```
834235973247071006940646759092387175730587641188932251116027038
380806185654011399021430699011171742042528719488468644367718086
1643245710284453484385719873524200530907393905143379094672667
2234643259349535186268571629077937597838801337973092285608744
2099515331998682280400044321325970733903633578923799976558788
5769633489221634507022764674985138120855404494044418286402651
370944982348959343901736635886964816823873508759380834448436
51362842197252338116053318150074245828908218872606828866325436
13109252862114326372077785369292570900594814481097443781269562
6473036714288957642240844022596051096003630989500919988913758
128395236132956672538139784348791727812172856528954691941812183
4307875450169474659873821524376974795657255598959459818063909
8344891175879455994652382137038240166358066403475457
e = 65537
d = 33410579727104715198165792151773083237115711907433085621783816
46770124086415914349868648088061339055175526760512412212208807
6644288306457909709901541778740781783644460982328809983396683
2697449469677098482618977788033303759560236815204203169682707
22475105090515003925555211936181996440875070997278483935982170
63018830414418427672760537663590125709985332601903139478800064
132117277409474470678034607060317611740874902282554360522445516
36823801781434029233115278710139434099796272520346062523994498
058074121550388611165999532399069719243650604162160473244358221
089118134071294983607455557349857985437381055885040711267092779
93864853519402750062690800513651147817777162022129559822144665
53104024367464797823689679431538303917047737574557738610501604
52401746790867834196061076588076120390140624579927639175546078
67721801124995486542353808003561237987405414200081219038791867
51398260888132027310556510513394033484402639310949836038576189
2328355263829025896714415424753631393699528207949577865467850
363091481887910300733053012017813368633412335162729677740578874
55945064156158502108102776915457729099888813405565995967337174
01089924014874029987229222036914331526326518454684517757259862
6122461295989845729128702403904222052115783265987057644225028
042582855101889823983060828368463449313329835894848802603093
48373229531127293270115611985532798401007828666273
```

```
# Step 1: Encrypt the message
message = "hello Marseene prime factorisation in practice"
m = bytes_to_long(message.encode())  # Convert message to integer
c = pow(m, e, n)  # Encrypt using c = m^e mod n
print(f"Ciphertext: {c}")

# Step 2: Decrypt the message
decrypted_m = pow(c, d, n)  # Decrypt using m = c^d mod n
decrypted_message = long_to_bytes(decrypted_m).decode(errors='ignore')
# Convert back   to string
print(f"Decrypted message: {decrypted_message}")
```

🎯 Attack Solution (Factorizing Mersenne Primes)

```
from Crypto.Util.number import long_to_bytes
from sage.all import *

# Given values
n = 65841627483018454412502751992144351578988826415607473309924
40401262136824977140327981163992881765024628292557845259777229
03018714434309698108208388664768262754316426220651576623731617
882923164117579624827261244506084274371250277849351631679441171
0184180184980399964725498931505771893028715203117151797307143121
8145624509784849166979599728983061298805852396838480882282837
09001984892492433991651252192447537907797644662369651357935765
161932131750614016673886222283620427170540146790329534410340215
068560170810626175723511954185058993887157097959920295590421197
834235973247071006940646759092387175730587641188932251116027038
380806185654011399021430699011171742042528719488468644367718086
1643245710284453484385719873524200530907393905143379094672667
2234643259349535186268571629077937597838801337973092285608744
2099515331998682280400044321325970733903633578923799976558788
5769633489221634507022764674985138120855404494044418286402651
```

```
370944982348959343901736635886964816823873508759380834448436
51362842197252338116053318150074245828908218872606828866325436
13109252862114326372077785369292570900594814481097443781269562
6473036714288957642240844022596051096003630989500919988913758
128395236132956672538139784348791727812172856528954691941812183
4307875450169474659873821524376974795657255598959459818063909
8344891175879455994652382137038240166358066403475457
e = 65537
c=50377012627219503541961321721967094168789194799359361160012568
2069319210728688802342879711596617238249513380591119617328591516
1674673990790652880235861203890953279961918256760536294463287
6077780800953979137492880483034945807185934548065370104767771
0440750986760338395995593877244914898700421804051735903100716
64776906998290821948479318260036925912847805187762455041125769
82411287003176295980932387496298008607517424579689078441934743
974631473184953419944544091074164311698202867087135804336812680
578613481249096156372869191176717819270194904537392033776868682
32293959616346868151220596077990722746576228163863518928431167
01308069821338767032816739141436920991886355768327749024555255
98293142362042700301741272153404609659764265712685997802061058
06565711320849762738390150976876099676926444150266146380887921
87303180561082672475787006169874705526448666671382940553900015
06010991851641806334751260329764289420901828162947054231050985
80390224630946916479055900125853630869079583111351556162902146
57708212431041474568227114807700566056784843737679875787483410
8065421898592314776076725405033398049124036627936040260848163
061349200074835506682922598951630826365739009630546690849252
71165399761025979284755165872923347036750832522592827661722814
911885183298636396137385136171063398022135765186092225776426064
5747382843410285709238935076518705947882144430419

# Step 1: Find small values of s1, s2 such that 2^s - 1 divides n
p, q = None, None
for s1 in range(100, 3000):
    p_candidate = (1 << s1) - 1
    if n % p_candidate == 0:
```

```
        q_candidate = n // p_candidate
        if q_candidate.bit_length() in range(100, 3000):  # Ensure q is also in the r
ange
            p, q = p_candidate, q_candidate
            print(f"Found factors: p={p}, q={q}")
            break

if not p or not q:
    raise ValueError("Failed to factorize n with Mersenne primes in given rang
e")

# Step 2: Compute φ(n)
phi_n = (p - 1) * (q - 1)

# Step 3: Compute d (modular inverse of e mod φ(n))
d = pow(e, -1, phi_n)

# Step 4: Decrypt the message
m = pow(c, d, n)



# Step 5: Convert back to bytes
flag = long_to_bytes(m)
print(f"Decrypted flag: {flag.decode(errors='ignore')}")
```

📋 Output

- **Decrypted Message**: Successfully recovers the original message by factorizing `n` and computing the private key `d`.

✅ Key Takeaways

- **RSA encryption can be broken** when the modulus `n` is a product of **Mersenne primes**, enabling factorization of `n`.

- **Mersenne prime factorization attacks** take advantage of the properties of these special primes to recover the private key.
- **Mitigation:** RSA keys should not be generated using Mersenne primes to avoid this specific vulnerability.