Graham Crackers Solution

Writeup

- **Author**: supasuge | Evan Pardon
- **Difficulty**: Hard

Challenge Source Code

Below is the source code for this challenge (created by me).

```
#!/usr/bin/env sage
from Crypto.Util.number import long_to_bytes, bytes_to_long
from sage.all import *
import os
length_N = 4096
Kbits = 200
e = 3
FLAG = next((open(f, 'rb').read().strip() for f in ['flag.txt', 'flag.example'] if
os.path.isfile(f)), b"GrizzCTF{FAKE_FLAG_LOL!!}")
K = bytes_to_long(FLAG)
if K.bit_length() > Kbits:
    raise ValueError("You done goofed... Think about what you have done.")
p = next_prime(2^(length_N//2))
q = next_prime(p)
N = p*q
M = (2^length_N) - (2^Kbits) + K
assert M < N
C = pow(M, e, N)
print(f"N = {N}")
```

```
print(f"e = {e}")
print(f"C = {C}")
print(f"length_N = {length_N}")
print(f"Kbits = {Kbits}")
HINT = """
[+]-------------------------------------------------------------------------------
--[+]
[+]                    Hint:
[+]
[+] - We constructed M as M = 2^length_N - 2^Kbits + K, where K encodes the flag.
[+]
[+] - Your goal: Given N, e, C, and the knowledge of M's structure, recover K (and thus the
flag).[+]
[+]-------------------------------------------------------------------------------
--[+]
"""
print(HINT)
```

RSA Refresher

In RSA, we have a public key (N, e) where:

- $N = p \times q$, with p and q being large primes
  Encryption:
- $C \equiv M^e \mod N$
  Decryption:
- $\phi(N) = (p - 1) \times (q - 1)$
  Then finding the private exponent d:
- $d \equiv e^{-1} \mod \phi(N)$
  Finally, recovering the plaintext:
- $M \equiv C^d \mod N$

Overview

Coppersmith's method, and in particular the Howgrave–Graham variant, is designed to efficiently find small roots of a univariate polynomial modulo a composite number N. In our context, we exploit the structure of M to recover the small unknown K.

Why this works

- **Small Root assumption**: CHG method works if there exists an integer root x_0 of a polynomial $f(x)$ with the property that $|x_0| < X$ for some bound X. Here, x_0 corresponds to K.
- **Known message structure**: Because M is constructed as a large number minus a shifted value plus a small K, the "error" introduced by K is small. This is the exact scenario where Howgrave–Graham can be applied.
- - **Lattice Techniques:** The method involves constructing a lattice from shifted and scaled versions of the polynomial $f(x)$ and then applying the LLL algorithm. If the lattice is chosen correctly, one of the reduced basis vectors will yield a new polynomial that has the same small root K over the integers.

Reference: For a detailed explanation of this technique and its application to RSA, see 20 Years of Attacks on RSA Cryptosystem by Boneh, which discusses various lattice-based attacks on RSA.

Mathematical Setup for the Challenge

Give the known structure of M, we define the polynomial:

$$f(x) = \left(2^{\text{length}_N} - 2^{K_{\text{bits}}} + x\right)^e - C.$$

Here, the unknown x represents K. Since $K \ll N$, we expect that the true root $x = K$ is small. The objective is to find this root such that:

$$f(K) \equiv 0 \mod N.$$

Lattice Construction

Arguably the most confusing part of this attack is properly constructing a set of polynomials derived from $f(x)$ to build a lattice. The general form of the lattice basis polynomials is:

$$g_{i,j}(x) = (xX)^j \cdot N^{m-i} \cdot (f(xX))^i$$

- m: Multiplicity
- t: Number of shifts
- X: An appropriate scaling factor related to the bound on K.

We define this lattice as follows:

- Given: $N, e, C, \text{and the form of} M = 2^{\text{length} N} - 2^{K\text{bits}} + K$
- Define $f(x) = (2^{\text{length}_N} - 2^{K_{\text{bits}}} + x)^e - C$
- Construct a lattice basis from polynomials using the general form defined above.

For $0 \leq i < m, 0 \leq j < d$, and similarly for the last t polynomials:

$$g_{\text{last},i}(x) = (xX)^i f(xX)^m$$

Form the lattice from the coefficients of these polynomials. After LLL reduction, we obtain a short vector that corresponds to a polynomial with an integer root, that root being our secret K .

SageMath Example from Solution

```
polZ = pol.change_ring(ZZ)
    x = polZ.parent().gen()
    print(f"\n{COLORS.YELLOW}Constructing lattice basis...{COLORS.RESET}")
    # build polynomials for lattice basis
    gg = []
```

```
    for ii in range(mm):
        for jj in range(dd):
            gg.append((x*XX)^jj * modulus^(mm - ii) * (polZ(x*XX))^ii)
    for ii in range(tt):
        gg.append((x*XX)^ii * (polZ(x*XX))^mm)

    nn = len(gg)
    BB = Matrix(ZZ, nn)

    for i in range(nn):
        for j in range(i+1):
            BB[i, j] = gg[i][j]
```

Challenge Context

In this challenge, we're provided:

- RSA modulus $N = pq$.
- Public exponent e.
- Ciphertext $C \equiv M^e \mod N$.
  The plaintext M structured as:

$$M = 2^{\text{length}_N} - 2^{K_{\text{bits}}} + K$$

where:

- length_N is the bit-length of N.
- K_{bits} is the bit-length of the unknown integer K.
- K encodes the flag.

Given that $K \ll N$, CHG is suitable for finding the integer K.

Parameter selection

- $e = 3$, as given in the challenge
- $K_{\text{bits}} = 200$
- M and t are chosen based off of theoretical bounds defined in the CHG method:

$$m = \left\lceil \frac{\beta^2}{d\epsilon} \right\rceil,$$

$$t = \left\lfloor dm \left(\frac{1}{\beta} - 1 \right) \right\rfloor$$

  where $\beta \approx 1$ and d is the degree of the polynomial:

- X is chosen as:

$$X \approx \lceil N^{(\beta^2/d)-\epsilon} \rceil$$

  The goal is for these conditions to satisfy the following inequality:

$$X^{n-1} < \frac{N^{\beta m}}{\sqrt{n}}$$

Where n is the dimension of the lattice. If this condition is met, Howgrave-Graham's theorem guarantees that the “hidden” small root can be extracted.

LLL Reduction & Root extraction

Once the lattice is constructed, the LLL algorithm is applied to find a reduced basis. Typically, one of the vectors in this reduced basis corresponds to a polynomial that, when reinterpreted in the original variable x, has the same small root K.

1. **Construct the lattice basis:** Using the polynomials $g_{i,j}(x)$, a basis matrix is built whose entries are the coefficients of these polynomials.
2. **Apply LLL Reduction:** The LLL algorithm is used to reduce the lattice. The hope is that the first vector in the reduced basis corresponds to a polynomial with small coefficients.
3. **Extract the Root:** The reduced polynomial is then solved over the integers. Because K is known to be small, the root extraction is straightforward.
4. **Reconstruct the Message:** Once K is recovered, reconstruct the plaintext M as:

$$M = 2^{\text{length}_N} - 2^{K_{\text{bits}}} + K.$$

   Converting M to its byte representation reveals the flag.

Reference: David Wong's repository on RSA and LLL Attacks provides practical examples and insights into constructing such lattices and performing LLL reduction.

Solution source code

```
#!/usr/bin/sage
from sage.all import *
import time
from Crypto.Util.number import long_to_bytes, bytes_to_long
import sys
from dataclasses import dataclass
# parameters from `out.txt`
@dataclass
class COLORS:
    RED = '\033[91m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
```

```
    BLUE = '\033[94m'
    MAGENTA = '\033[95m'
    CYAN = '\033[96m'
    BOLD = '\033[1m'
    RESET = '\033[0m'

def print_mathematical_context(N, e, C, length_N, Kbits):
    print_header("Mathematical Context")
    print(f"""
    In the Howgrave-Graham variant of Coppersmith's method, we're solving:

    $M = 2^{{length_N}} - 2^{{Kbits}} + K$ (where K is small)
    $C ≡ M^e \\pmod{{N}}$

    Therefore:
    $C ≡ (2^{{length_N}} - 2^{{Kbits}} + K)^e \\pmod{{N}}$

    Let's define our polynomial:
    $f(x) = (2^{{length_N}} - 2^{{Kbits}} + x)^e - C$
    """)
    print(f"\n{COLORS.YELLOW}Parameters:{COLORS.RESET}")
    print(f"N: {N} (bit length: {N.nbits()})")
    print(f"e: {e}")
    print(f"length_N: {length_N}")
    print(f"Kbits: {Kbits}")

def explain_beta_choice(beta, N, Kbits):
    print_header("Beta Selection Analysis")
    print(f"""
    The parameter β (beta) determines the bound on our solution:
    - We want to find roots < $N^β$
    - In our case, K is approximately $2^{{Kbits}}$
    - Therefore, we need: $2^{{Kbits}} < N^β$
    """)
    print(f"\n{COLORS.YELLOW}In this case:{COLORS.RESET}")
    print(f"$2^{{{Kbits}}} < N^{{{beta}}}$")
```

```
    print(f"$2^{{{Kbits}}} < 2^{{{N.nbits()}·{beta}}}$")
    print(f"Required β > {Kbits/N.nbits():.4f}")
    print(f"Chosen β = {beta}")
```

```
N =
10443888814131525066917527107166243825799642490473837803842334832839539079715574568488268119349975583408901067144392628379875734381857936072632360878513652779459569765437099983403615901343837183144280700118559462263763188393977127456723346843445866174968079087058037040712840487401186091144679777835980290066869389768817877859469056301902609405995794534328234693030266964430590250159723998677142155416938355598852914863182379144344967340878118726394964751001890413490084170616750936683338505510329720882695507699836163694119330152137968258371880918336567512213184928463681255502259983004123447848625956744921946171077660876865116077929890850172521438153366130061185257177878064383870607259742656204479656865068145537880732773822501901057054051863750245568332910068237736379832922845277794012167967098358506905213810135832024483707991861564895221962382480398017068899840615985101435519935645433785052921430521788268872785126417810027990840568395363972762173111095691900450377667859350990328055178583545209800902741285878085041798115310370248482060671012478979062125527724769651395475928463357133457925750611607159634526367834290690830925884785512842165479869356846932253874689515643470416444714581891536991170971019123898732341630209011
e = 3
C =
10443888814131525066917527107166243825799642490473837803842334832839539079715574568488268119349975583408901067144392628379875734381857936072632360878513652779459569765437099983403615901343837183144280700118559462263763188393977127456723346843445866174968079087058037040712840487401186091144679777835980290066869389768817877859469056301902609405995794534328234693030266964430590250159723998677142155416938355598852914863182379144344967340878118726394964751001890413490084170616750936683338505506949442702736844580838861445726134402239652540396374475443644152597423724667516490911757590637324130593922955500238255740310755007639901776948220037281360703599092706426514890204913775712291257668550842094923478960800741150734129739245853212958672635300429258826854201618145115142017765943743782526587106244696899564843317788064971143140750140391339085403174107593735013830831475614000711468958055785365511911643453863835178653217561911672189107260434254087513393704984995753447611313758507341556536611332749055288441754148211506457580380267002896343211548125652974288311492704186185302573818460340861899335942643120078501720282340482802804005033540968583993703397392815390308712495496907555886391574881727977411476772804636667026477781013651
```

```
length_N = 4096
Kbits = 200

def print_header(text):
    print(f"{COLORS.BLUE}{'='*50}{COLORS.RESET}")
    print(f"{COLORS.BOLD}{text}{COLORS.RESET}")
    print(f"{COLORS.BLUE}{'='*50}{COLORS.RESET}")

def matrix_overview(BB, bound):
    dims = BB.dimensions()
    print(f"\n{COLORS.YELLOW}Lattice Matrix Overview{COLORS.RESET}: {dims[0]} x {dims[1]}")
    print(f"\n{COLORS.YELLOW}Matrix Structure{COLORS.RESET} (X=non-zero, 0=zero):")
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print(a)

def coppersmith_howgrave_univariate(pol, modulus, beta, mm, tt, XX):
    dd = pol.degree()
    nn = dd * mm + tt
    print(f"\n{COLORS.YELLOW}Coppersmith Parameters:{COLORS.RESET}")
    print(f"d (polynomial degree): {dd}")
    print(f"m (multiplicity): {mm}")
    print(f"t (extra shifts): {tt}")
    print(f"X (bound): {XX}")
    print("""
    Howgrave-Graham's Theorem states that we need:
    $\\|p(xX)\\| < \\frac{N^m}{\\sqrt{n}}$ where $n$ is the lattice dimension
    """)
    if not 0 < beta <= 1:
        raise ValueError("beta should be in (0,1]")
    if not pol.is_monic():
```

```
        raise ArithmeticError("Polynomial must be monic.")


    # sanity debug print
    print(f"\n{COLORS.YELLOW}Checking Howgrave-Graham Conditions:{COLORS.RESET}")
    cond1 = RR(XX^(nn-1))
    cond2 = pow(modulus, beta*mm)
    print(f"$X^{{n-1}} = {cond1}$")
    print(f"$N^{{β·m}} = {cond2}$")
    print(f"Condition satisfied: {COLORS.GREEN if cond1 < cond2 else COLORS.RED}{cond1 < cond2}
{COLORS.RESET}")


    polZ = pol.change_ring(ZZ)
    x = polZ.parent().gen()
    print(f"\n{COLORS.YELLOW}Constructing lattice basis...{COLORS.RESET}")
    # build polynomials for lattice basis
    gg = []
    for ii in range(mm):
        for jj in range(dd):
            gg.append((x*XX)^jj * modulus^(mm - ii) * (polZ(x*XX))^ii)
    for ii in range(tt):
        gg.append((x*XX)^ii * (polZ(x*XX))^mm)

    nn = len(gg)
    BB = Matrix(ZZ, nn)

    for i in range(nn):
        for j in range(i+1):
            BB[i, j] = gg[i][j]

    print(f"\n{COLORS.YELLOW}Initial Lattice Matrix:{COLORS.RESET}")
    matrix_overview(BB, modulus^mm)

    print(f"\n{COLORS.YELLOW}Applying LLL reduction...{COLORS.RESET}")
    BB = BB.LLL()
```

```
    # construct new polynomial
    print(f"\n{COLORS.YELLOW}Constructing polynomial from first LLL vector...{COLORS.RESET}")
    new_pol = 0
    for i in range(nn):
        new_pol += x^i * BB[0, i]/XX^i

    # factor polynomial and check roots
    potential_roots = new_pol.roots()
    print(f"\n{COLORS.YELLOW}Potential roots found:{COLORS.RESET} {potential_roots}")
    roots = []
    polZ = polZ.change_ring(ZZ)
    for root in potential_roots:
        rr = root[0]
        if rr.is_integer():
            val = polZ(ZZ(rr))
            if gcd(modulus, val) >= modulus^beta:
                roots.append(ZZ(rr))
    return roots


ZmodN = Zmod(N)
P.<x> = PolynomialRing(ZmodN)
# Polynomial from the original example:
# pol = (2^length_N - 2^Kbits + x)^e - C
print_mathematical_context(N, e, C, length_N, Kbits)
# our polynomial
pol = (2^length_N - 2^Kbits + x)^e - C
dd = pol.degree()
print(f"degree of polynomial: {dd}")
# Parameters for Coppersmith
beta = 1   # b = N #
epsilon = beta / 7
mm = ceil(beta^2 / (dd * epsilon))
tt = floor(dd * mm * ((1/beta) - 1))
XX = ceil(N^((beta^2/dd) - epsilon))
```

```
start_time = time.perf_counter()
try:
    roots = coppersmith_howgrave_univariate(pol, N, beta, mm, tt, XX)
    if len(roots) > 0:
        K = roots[0]
        print(f"\n{COLORS.GREEN}Found K:{COLORS.RESET} {K}")
        M = 2^length_N - 2^Kbits + K
        M_bytes = long_to_bytes(M)
        try:
            msg = M_bytes.decode('utf-8', 'ignore')
            print(f"\n{COLORS.GREEN}Recovered flag:{COLORS.RESET} {msg}")
        except:
            print(f"\n{COLORS.RED}Could not decode message directly. Raw bytes:{COLORS.RESET}",
M_bytes)
    else:
        print(f"\n{COLORS.RED}No roots found.{COLORS.RESET}")
except Exception as e:
    print(f"\n{COLORS.RED}Error occurred:{COLORS.RESET} {e}")
finally:
    elapsed = time.perf_counter() - start_time
    print(f"\n{COLORS.BLUE}Time taken:{COLORS.RESET} {elapsed:.2f} seconds")
```

Conclusion

This challenge demonstrates the practical application of the CHG method for attacking structured RSA plaintexts. Through careful mathematical formulation, lattice reduction, and small-root finding, we successfully retrieved the flag:

```
GrizzCTF{Graham_Cracked!}
```

Sample Run (`solve.sage`)

Below is the output from running `solve.sage`

In the Howgrave-Graham variant of Coppersmith's method, we're solving:

$M = 2^{length_N} - 2^{Kbits} + K$ (where K is small)
$C \equiv M^e \pmod{N}$

Therefore:
$C \equiv (2^{length_N} - 2^{Kbits} + K)^e \pmod{N}$

Let's define our polynomial:
$f(x) = (2^{length_N} - 2^{Kbits} + x)^e - C$

Parameters:
N:
$104438888141315250669175271071662438257996424904738378038423348328395390797155745684882681193499755834089$
$\text{(bit length: 4097)}$
e: 3
$\text{N bit length: 4096}$
Kbits: 200
Coppersmith Parameters:
d (polynomial degree): 3
m (multiplicity): 3
t (extra shifts): 0
X (bound):
$725664144228083523330177862176037906395366581733741235382516029717794707172769935060531133995664891609271$

Howgrave-Graham's Theorem states that we need:

$$\|p(xX)\| < \frac{N^m}{\sqrt{n}}$$

where n is the lattice dimension

Checking Howgrave-Graham Conditions:
$X^{n-1} = 7.68927365891604e1878$
$N^{\beta \cdot m} = 11391652252630433708459385793159320086212984680286145011137995334266424391724696915673064777116370$
Condition satisfied: True
Constructing lattice basis...

Initial Lattice Matrix:

- Lattice Matrix Overview: 9 x 9

```
Matrix Structure (X=non-zero, 0=zero):
00 X 0 0 0 0 0 0 0 0 ~
01 0 X 0 0 0 0 0 0 0 ~
02 0 0 X 0 0 0 0 0 0 ~
03 X X X X 0 0 0 0 0
04 0 X X X X 0 0 0 0
05 0 0 X X X X 0 0 0
06 X X X X X X X 0 0
07 0 X X X X X X X 0
08 0 0 X X X X X X X
```

Applying LLL reduction...

Constructing polynomial from first LLL vector...

Potential roots found: $(4484795979052873665628013372961328190161158652140990824533733, 2)$

Found K: $4484795979052873665628013372961328190161158652140990824533733$

Recovered flag: `GrizzCTF{Graham_Cracked!}`

Time taken: 0.14 seconds

References

- David Wong - RSA and LLL Attacks
    - YouTube Explanation
- 20 Years of Attacks on RSA Cryptosystem