CS 355 Lecture #4: Attacking deployed RSA.
Quick recap: Random Oracles.

The Random Oracle Model (Bellare & Rogaway, CCS '93) is a pragmatic tool for building cryptosystems.

But: security is heuristic when we instantiate!

<table>
<thead>
<tr>
<th>ROM world</th>
<th>Real world</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H(x)$ is a public random function</td>
<td>$H(x) \triangleq \text{SHA3}(x)$</td>
</tr>
<tr>
<td>Can prove properties of protocols that call $H(x)$</td>
<td>Must assume that no adversary can exploit the structure of SHA3.</td>
</tr>
</tbody>
</table>
Today: Attacking real-world deployments.

MSB: the problem is usually not "the crypto"

1. Reminder: RSA
2. Mining your P's & Q's → randomness failure
3. Return of Coppersmith's Attack → "optimization" = ouch
Let $N$ be an integer with (secret) factorization

$$N = pq .$$

$PK \triangleq (N, e)$

$SK \triangleq (p, q)$

$OR \quad d = e^{-1} \mod (p-1)(q-1)$

Coprime to $(p-1)(q-1) \Rightarrow$ usually $2^{16}+1$
Recall: PK defines a permutation on \( \mathbb{Z} \mod N \):

\[ x \mapsto x^e \mod N \]

Without SK, this seems hard to invert for big \( N \) (think: \( N \) is \( >2048 \) bits).

Given SK, inverse is

\[ y \mapsto y^d \mod N \]

\( \mathbb{Z}_N^* \) has order \( \varphi(N) = (p-1)(q-1) \)

So \( (x^e)^d \equiv x \mod N \).
RSA "trapdoor" permutation gives:

→ Encryption (e.g. RSA-OAEP\textsuperscript{+}
due to Shoup '01)

→ Signatures (e.g. RSA-PSS
due to Bellare & Rogaway '96)

Never use the RSA permutation directly for encryption or signing!

\[\text{this is the most important item in today's lecture.}\]
2. **Mining your Ps & Qs**

- Heninger, Durumeric, Wustrow, Halderman
  \[ \Rightarrow \text{USENIX Security 2012} \]
- Also: Lenstra, Hughes, Augier Bos, Kleinjung, Wachter 2012
  "Ron was wrong, Whit is right"
  \[ \Rightarrow \text{ePrint #2012/064} \]

Idea: if two RSA moduli \( N_1 \neq N_2 \) have nontrivial \( \text{GCD} \), we can factor both.

\[
\begin{align*}
N_1 &= P \cdot q_1 \\
N_2 &= P \cdot q_2
\end{align*}
\]

\( \text{GCD}(N_1, N_2) = P \)

\( \Rightarrow \text{now divide} \)
Strategy:

1. Collect millions of RSA PKs
2. Check for nontrivial GCDs
3. Profit?

Step 1: Scan entire IPv4 address range for TLS certificates
Zakir’s software, Zmap, does this in minutes!

OR, maybe: collect RSA pub keys from GitHub?
⇒ millions are available...
Step 2: How do we check \( \approx 2^{23} \) keys for pairwise GCDs?

→ Naïvely, \( 2^{46} \) pairs of keys \( \Rightarrow 30 \text{ CPU-years} \)

L already OK for NSA.

→ Better: GCD tree

Daniel Bernstein, "How to find smooth parts of integers."

- GCD costs \( \tilde{O}(n) \) for \( n \)-bit inputs
- For \( k \) keys, naive cost is \( \tilde{O}(k^2 n) \)
- GCD tree reduces cost to \( \tilde{O}(kn) \)

Concretely: years become days.
GCD tree idea:

(1) Compute \( \prod_{i=1}^{k} N_k \)

\( \Rightarrow \) using a tree!

\( N_1 \quad N_2 \quad N_3 \quad N_4 \)

\( N_1 N_2 \quad N_3 N_4 \)

\( N_1 N_2 N_3 N_4 \)

\( \Rightarrow \) total cost \( \tilde{O}(kn) \)

(2) Compute \( r_1 = \Pi \mod N_1 \)
   (and so forth)

(3) GCD \( \left( \frac{r_1}{N_1}, N_1 \right) \) gives
    a common factor between \( N_1 \) & some other \( N_i \) (etc.)
Why does (3) work?

\[ \pi = N_1 N_2 N_3 N_4 \]

Assume \( N_1 = p_1 q_1, \ N_2 = p_2 q_2, \) no other common factors

\[ \pi = (p_1 q_1)(p_2 q_2) \times N_3 N_4 \]

\[ \Rightarrow \pi = \pi \mod N_1^2 \]

\[ = p^2 q_1 q_2 \text{(garbage)} \mod N_1^2 \]

\[ \Rightarrow \frac{r_1}{N_1} = \frac{p_2 q_2 \text{(garbage)}}{N_1} \mod N_1^2 \]

\[ = p_2 q_2 \text{(garbage)} \]

\[ \Rightarrow \text{GCD} \left( \frac{r_1}{N_1}, N_1 \right) \]

\[ = \text{GCD} \left( p_2 q_2 \text{(garbage)}, p_1 q_1 \right) \]

\[ = p \overset{?}{=} \text{got it!} \]
How do we compute $r_i$?

Naively, each $\pi \mod N_i$ costs $\mathcal{O}(kn)$.

$\Rightarrow \pi$ is $kn$ bits!

$\Rightarrow$ Computing all $r_i$ costs $\mathcal{O}(k^2n)$.

Better: a tree!

$$\pi = N_1 N_2 N_3 N_4$$

$\pi \mod (N_1 N_2)^2$

$\pi \mod N_1^2 = r_1$

$\pi \mod N_2^2 = r_2$

$\pi \mod (N_3 N_4)^2$

$\pi \mod N_3^2 = r_3$

$\pi \mod N_4^2 = r_4$

$\Rightarrow$ values shrink at each step

$\Rightarrow \tilde{\mathcal{O}}(kn)$ cost in total!
Heninger et al. factored >64,000 keys that they found in the wild!

→ WHY ARE KEYS SO BAD?

→ IBM made devices that chose from a list of 9 possible prime factors

→ Embedded systems often have trouble generating "good" randomness

→ at boot, devices generate keys before they have gathered sufficient entropy

→ see also: Debian RNG fiasco 2008.
Return of Coppersmith's Attack
Nemec, Sys, Svenda, Klinec, Matyas
ACM CCS 2017

Idea: for RSA moduli $N = pq$ where $p$ & $q$ have special structure, we can factor $N$. 

Why special structure?

$\Rightarrow$ generate keys using fewer random bits on embedded devices (made by Infineon)

Result: millions of devices were recalled!
The special structure:
\[ p = k \cdot M + (65537^a \mod M) \]
\[ q = l \cdot M + (65537^b \mod M) \]
for a public constant \( M \), a primorial — the product of the first \( j \) primes.

Aside: why this choice of \( M \)?

\[ \Rightarrow \text{guarantees } p \& q \text{ are not divisible by small primes} \]
\[ \Rightarrow \text{fewer primality tests, so faster key generation} \]
\[ \Rightarrow \text{a tragic optimization} \]
Intuition leads us astray:

\[ p = k \cdot M + (65537^a \mod M) \]

for 1024-bit \( p, M \) is "\( p^{126} \)" — 971 bits.

\[ k \text{ has } \approx 53 \text{ bits, } a \text{ has } > 100 \text{ bits} \]

> \( > 2^{128} \) choices for \( p \)

→ What could go wrong?

Coppersmith's algorithm (Eurocrypt '96) lets us recover \( p, q \) from \( N \) in polynomial time if we know \( > 1/2 \) the bits of either!
Theorem (Coppersmith '96): Let \( N = pq \) be an RSA modulus. Let \( f \in \mathbb{Z}_N[x] \) be a polynomial of degree \( d \). Then we can find all integers \( x_0 \) s.t. \( f(x_0) = 0 \mod p \) where \( |x_0| \leq N^{1/4d} \) in time polynomial in \( d \) and \( \log N \).

Note: Since we find all such solutions, there can only be \( \text{poly}(d, \log N) \) of them.
Recall: \( p = KM + (65537^a \mod M) \)

**Attack**: (1) guess \( a \)
(2) recover \( k \) w/ Coppersmith.

**Step (2) first**: given \( a \),
\[
p = C_1 \cdot k + C_2
\]
\[
f(x) \equiv C_1 x + C_2
\]
\[
\Rightarrow f(k) = p \equiv 0 \mod p
\]

Coppersmith? \( \deg(f) = 1 \)

So we will get candidate \( k \) values up to \( N^{1/4} \). \( p \) is 1024 bits, \( M \) is 971 bits \( \Rightarrow \) real \( k \approx N^{1/4} \)

For each candidate \( k \), try factoring \( N \). (We know there won't be too many...)
Step (1): Guessing $a$'s value.

Really, guess $C_2 = 65537^a \mod M$ \( \Rightarrow \) the constant term of \( f(x) \).

How many values of $C_2$ are there?

Equivalently: what is the size of the subgroup of $\mathbb{Z}_M^*$ generated by 65537?

\( \Rightarrow \) If $M$ were prime, could be as large as $M-1$.\( ^{\text{many small factors}} \)

\( \Rightarrow \) But $M$ is smooth, so the subgroup is small (and its size is easy to compute).

So: compute size of subgroup $r$, then "guess" $0 \leq a < r$ and run step (2).
Optimizing the attack:

- $M$ is "too big": Coppersmith gives solutions up to $N^{1/4}$, but we only need $N_{53}$-bit $k$.

Idea: pick $M'$ dividing $M$

S.t. $1024 - \log_2(M') \leq N^{1/4}$

$\Rightarrow$ now $p = k'M' + 65537^a \mod M'$

order of 65537 is smaller in $\mathbb{Z}_{M'}^*$ than in $\mathbb{Z}_M^*$ — fewer guesses

Trade-off: bigger $M'$ makes Coppersmith faster but takes more guesses for $C_2$.

$\Rightarrow$ optimize?
## Results

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Cost to break on AWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 bits</td>
<td>40 minutes, 6.3¢</td>
</tr>
<tr>
<td>1024 bits</td>
<td>32 days, $76</td>
</tr>
<tr>
<td>2048 bits</td>
<td>46 years, $40k</td>
</tr>
<tr>
<td>3072 bits</td>
<td>$10^{28}$ years, $3 \times 10^{11}$</td>
</tr>
<tr>
<td>4096 bits</td>
<td>$4 \times 10^8$ years, $3 \times 10^{11}$</td>
</tr>
</tbody>
</table>

Note: attack is trivially parallelized? (How?)
Identifying bad moduli

Recall:

\[ p = k \cdot M + (65537^a \mod M) \]
\[ q = l \cdot M + (65537^b \mod M) \]

So \[ N = pq \equiv 65537^{a+b} \mod M \]

A random RSA modulus has a vanishingly small chance of having this form \(- \ll 2^{-100}\)

So: check if \( N \mod M \) has a discrete log to the base 65537.

This is easy because \( M \) is smooth (many small factors).

Next lecture: discrete log!