UOSEC Week 2: Asymmetric Cryptography

Frank farana@uoregon.edu
IRC kee

Adam pond2@uoregon.edu
IRC xe0
IRC: irc.freenode.net #0x4f
Agenda

- HackIM CTF Results
- GITSC CTF this Saturday 10:00am
- Basics of Asymmetric Encryption
- Crypto Math
- Implementation
Why Asymmetric vs Symmetric?

-Symmetric functions require both parties to share a secret key to encrypt and decrypt messages, but transmitting a secret key can be tricky.

-Asymmetric functions attempt to solve this issue by using a private and public key pair.
Why Asymmetric vs Symmetric?

-Symmetric encryption follows the functions:
P = D(K, E(K, P))

-Asymmetric encryption looks like this:
P = D(K_S, E(K_p, P)), and P = D(K_p, E(K_S, P))

Where P = plaintext, K = key, K_p = public key, K_s = secret key, D() = decryption, and E() = Encryption. Also, C = E(), where C = ciphertext
Why Asymmetric vs Symmetric?

-The asymmetric methods for encryption and decryption mean that two different keys are needed, and only one needs to be kept a secret (the decryption key).

-Encryption keys (or public keys) allow Alice and Bob to send each other messages without risking their private keys needed for decryption.
Public Keys

Alice needs Bob’s credit card information (She sells scented candles) so she sends Bob her public key.

Alice

Alice’s $K_p$
5f4dcc3b5aa765d61
d8327deb882cf99

Bob
Public Keys

Alice needs Bob’s credit card information (She sells scented candles) so she sends Bob her public key.

Alice

| Alice’s $K_p$
| 5f4dcc3b5aa765d61d8327deb882cf99 |

Bob

| Alice’s $K_p$
| 5f4dcc3b5aa765d61d8327deb882cf99 |

Eve is up to her old ways and catches Alice’s key.
Public Keys

Bob uses Alice’s public key to encrypt his card information. \( E(K_p, P) = C \)

Alice

\( 5f4dcc3b5aa765d61d8327deb882cf99 \)

Eve

Bob

\( 5ebe2294ecd0e0f08eab7690d2a6ee69 \)

Ciphertext
Public Keys

Eve intercepts the ciphertext, and plans to decrypt it using $D(K_p, C)$.
Alice uses her secret key $K_S$ to decrypt the ciphertext back into plaintext, while Eve realizes something is not right...
Eve learns that this may not be so easy.

ONE DOES NOT SIMPLY SSH INTO MORDOR

SAURON USES PUBLIC KEY ENCRYPTION
Sending/Receiving Public Keys

-Send public keys as an email attachment, post them on your web page, print out QR codes for your friends, etc.

-Public keys eliminate a great deal of issues when it comes to sharing encryption keys.

-There are still some key management issues that cannot be solved with public keys.
Key Management Issues

- How do we know that Alice’s public key really belongs to Alice?

- If anyone can access a public key, how can one verify that the sender is who they claim to be?

- The Web of Trust (WoT) and certificate authorities can help.
Web of Trust

- Trust a third party entity to host public keys. (i.e. pgp.mit.edu)

- Others can ‘digitally sign’ your public key to add credibility.

- The authenticity is only as good as the word of the people backing it.
How Asymmetric Crypto Generally Works

- $E()$ and $D()$ are mathematical, one-way, inverse functions of one another. (That is if $E(x) = y$, then $D(y) = x$.)

- By definition, a one-way function is a simple to compute function, but extremely difficult to invert when given the output. ($E(x) = y$ is simple, but figuring out $x$ when given $y$ is difficult.)
Just “difficult” to compute?

-Encryption algorithms are essentially really hard math problems that would take a computer an unreasonably large amount of time to determine. (Thousands of years in some cases.)

-Based on Moore’s law, modern encryption will fail against the computers of the future. Just like encryption methods of the past fail today. (Try using WEP to protect your wireless router some time.)
Prime Number Generation

-Using large prime numbers strengthens the cryptographic equation, but how do we find such large prime numbers?

-Proving the Riemann Hypothesis would help (If you do that, let me know.)
Checking for Primality

-For a number $n$, checking numbers $2 - \sqrt{n}$ for factors can prove primality, but that will take forever.

-Are there other ways to check for primality?
Fermat’s Primality Test

Fermat came up with a test to determine if a number is prime:
\[ a^{P-1} = 1 \mod P, \text{ where } P = \text{prime number, and } 1 \leq a < P. \]

Basically for any \( a \),
\[ a^{P-1} \mod P = 1 \]

Otherwise the number has factors.
Fermat’s Primality Test Examples

P = 5
\[ a^4 \mod 5 = 1 \]
(1)\(^4 : 1 \mod 5 = 1 \checkmark 
(2)\(^4 : 16 \mod 5 = 1 \checkmark 
(3)\(^4 : 81 \mod 5 = 1 \checkmark 
(4)\(^4 : 256 \mod 5 = 1 \checkmark 

5 is prime.

P = 9
\[ a^8 \mod 6 = 1 \]
(1)\(^8 = 1 \mod 9 = 1 \checkmark 
(2)\(^8 = 256 \mod 9 = 4 \times 
(3)\(^8 = 6561 \mod 9 = 0 \times 
(4)\(^8 = 65536 \mod 9 = 7 \times 

9 is composite.
Exercise: Is this number prime?

563?
Exercise: Is this number prime?

563?
Yes!
Exercise: Is this number prime?

563?
Yes!

561?
Exercise: Is this number prime?

563?
Yes!

561?
It passes Fermat’s test, but 561 is not prime. Some numbers can appear to be prime, but aren’t. (Carmichael numbers)
Primality Tests

-Known tests can only give a probability of a number being prime. These are less costly to perform than a factoring algorithm.

-Miller Rabin Test, Fermat Primality Test

-Repeatedly testing a number can bring the likelihood of it being prime into a high percentage (~99%)
Carmichael Numbers

-Carmichael numbers can pass Fermat’s primality test, but are not actually prime.

-Using a non-prime number can severely reduce the complexity of an encryption algorithm.
Discrete Log Problem

Consider $3^x = 94143178827$
Discrete Log Problem

Consider $3^x = 94143178827$

This can quickly be solved with $\ln(3^x) = \ln(94143178827)$

$x = 23$
Discrete Log Problem

- Consider $3^x \mod 7 = 5$, where $x$ is an integer.

- Finding $x$ would take a brute force attempt, which isn’t too bad for mod 7, but consider:
  
  $12^x \mod 104729 = 100291$
  
  (12 is primitive root modulo 104729)

- The time complexity easily starts to get out of hand.
A CRYPTO NERD'S IMAGINATION:

His laptop's encrypted. Let's build a million-dollar cluster to crack it.

No good! It's 4096-bit RSA!

Blast! Our evil plan is foiled!

WHAT WOULD ACTUALLY HAPPEN:

His laptop's encrypted. Drug him and hit him with this $5 wrench until he tells us the password.

Got it.
Some Asymmetric Implementations

-Diffie-Hellman Key Exchange (1976)

-RSA (Rivest, Shamir, Adelman 1977)

-ElGamal (1985)

-PGP (Pretty Good Privacy 1991)
Diffie-Hellman Key Exchange

-What if Alice and Bob can share a secret key without worrying about Eve?

-Provides a public means of key distribution.

-Allows an $n$ participants to share a private key at one time.
Diffie-Hellman Algorithm

\[ y = g^x \mod P \]

\[ P = \text{publicly agreed prime} \]
\[ g = \text{a primitive root modulo } P \]
\[ x, y = \text{either a public or private key (depending on the phase)} \]
Diffie-Hellman Algorithm

\[ y = 5^x \mod 23 \]

A public function is used. It does not matter if this function is intercepted.
Diffie-Hellman Algorithm

\[ y = 5^x \mod 23 \]

Alice: \( a^s = 6 \)
Bob: \( b^s = 15 \)

Alice and Bob choose secret numbers. Keys \( a^s \) and \( b^s \) are never shared.
Diffie-Hellman Algorithm

\[ y = 5^x \mod 23 \]
\[ a^s = 6, \ b^s = 15 \]

Alice: \( a^p = 5^6 \mod 23 = 8 \)
Bob: \( b^p = 5^{15} \mod 23 = 19 \)

Alice and Bob generate their public keys independent from one another.
Once created, Alice will share \( a^p \) and Bob will share \( b^p \).
y = 5^x \mod 23
a^s = 6, \ b^s = 15
a^p = 8, \ b^p = 19
Alice: \ SS = 19^6 \mod 23 = 2
Bob: \ SS = 8^{15} \mod 23 = 2

The shared secret will be computed separately by both Alice and Bob when they use each other’s public in place of g.

2 will be their shared secret.
<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
<th>Eve</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = 5^x \mod 23$</td>
<td>$y = 5^x \mod 23$</td>
<td>$y = 5^x \mod 23$</td>
</tr>
<tr>
<td>$a^s = 6$</td>
<td>$b^s = 15$</td>
<td>$a^p = 8$, $b^p = 19$</td>
</tr>
<tr>
<td>$a^p = 8$, $b^p = 19$</td>
<td>$a^p = 8$, $b^p = 19$</td>
<td>$a^p = 8$, $b^p = 19$</td>
</tr>
<tr>
<td>SS = 2</td>
<td>SS = 2</td>
<td>SS = 2</td>
</tr>
</tbody>
</table>
Eve is left with some math problems...

$8 = 5^a \mod 23$

$19 = 5^b \mod 23$

$SS = 19^a \mod 23$

$SS = 8^b \mod 23$

Look familiar?
The discrete log problem strikes again.
Diffie-Hellman Weaknesses

- New keys must be generated for every group of users.
- Cannot be used to sign certificates
- Issues when implemented in OpenSSL
RSA

- One public key can be shared with as many people as needed.

- No need to manage multiple secret keys.
How RSA Works

Encryption: $c = m^e \mod n$

Decryption: $m = c^d \mod n$

Where

$n,e$ are the public key
$d$ is the private key
$m$ is plaintext, $c$ is ciphertext
RSA Algorithm

’Randomly’ generate two distinct prime numbers P and Q, with similar bit length.

- \( n = PQ \) (This will be mod)

- Compute \( \varphi(n) \)

- Choose e such that e and \( \varphi(n) \) are coprime. (shorter bit length is typically more efficient). e is the public key exponent.

- Find d where \( de = 1 \mod \varphi(n) \), d is the private key exponent.
Coprime? \( \varphi(n) \)?

- Coprime numbers have a gcd of 1. (Example: 6 and 35)

- Euler's totient function \( \varphi(n) \) will give the amount of numbers 1-n that are not factors of n.

\( \varphi(n) \) is also multiplicative. \( \varphi(j \times k) = \varphi(j) \times \varphi(k) \)

Examples:

\( \varphi(9) = 6 \) (That is 2,4,5,6,7,8)

\( \varphi(11) = 10 \) (Phi of a prime number \( \varphi(P) \) will always be \( P-1 \))
RSA Example

Step 1: Select primes P,Q
P = 61
Q = 53

Step 2: Compute n (PQ)
3233
RSA Example

Step 3: Find $\varphi(n)$.
Since $n = PQ$, $\varphi(PQ) = \varphi(P) \times \varphi(Q) = \varphi(n)$.
$\varphi(3233) = \varphi(61) \times \varphi(53) = (60)(52) = 3120$

Step 4: Find an $e$ coprime with 3120, and $1 < e < 3120$.
e = 17 (Smaller bit length is usually more efficient without sacrificing complexity.)
RSA Example

Step 5: Find $d$ where $d(17) = 1 \mod 3120$. (The modular multiplicative inverse)

$d \ (17) \mod 3120 = 1$
$2753(17) \mod 3120 = 1$
$46801 \mod 3120 = 1$
$d = 2753$
RSA Example

We have our Encryption and Decryption functions!

Encryption: \( c = m^e \mod n \) > \( c = m^{17} \mod 3233 \)

Decryption: \( m = c^d \mod n \) > \( m = c^{2753} \mod 3233 \)
Almost there...

Let’s encrypt an ascii “A” which is 65.

c = (65)^{17} \mod 3233 = 2790
Almost there...

Finally, let’s decrypt 2790

\[ m = (2790)^{2753} \mod 3233 = 65 \]
RSA Weaknesses

- Timing attacks (Listening to the CPU)
- Too small of an e compared to the modulus
- Chinese Remainder Theorem
Digital signatures in RSA

Message hash = h, Signature = s

Signing: $h_1^d \mod n = s$

Verification: $s^e \mod n = h_2$

If $h_1 = h_2$ the message is genuine and not tampered with.
Exercise: Brute Force RSA!

I secretly encoded a message using RSA. I want you to decrypt it for me using only the following information:

\[ e = 5 \]
\[ N = 10142789312725007 \]
\[ c = 2010448083501228 \]

helper files and copyable text at:
http://uosec.net/week2/
Exercise: Brute Force RSA!

e = 5
N = 10142789312725007
c = 2010448083501228

hint: d*e \equiv 1 \mod \phi(N)

helper files and copyable text at:
http://uosec.net/week2/
Exercise: Brute Force RSA!

e = 5
N = 10142789312725007
c = 2010448083501228

hint: d*e ≡ 1 mod φ(N)

hint2: m = c^d mod N

helper files and copyable text at:
http://uosec.net/week2/
Exercise: Brute Force RSA!

Answer: RSA_FLAG
Exercise: Brute Force RSA!

Answer: RSA_FLAG

Steps to solve:
1. Factor 10142789312725007 using helper.py or wolfram thus you get $p = 100711409$ $q = 100711423$
Exercise: Brute Force RSA!

Answer: RSA_FLAG

Steps to solve:

1. Factor 10142789312725007 using helper.py or wolfram
   thus you get p = 100711409 q = 100711423

2. Obtain decryption exponent d by solving
   \[ d = e^{-1} \mod \phi(N) = 8114231289041741 \]
Exercise: Brute Force RSA!

Answer: RSA_FLAG

Steps to solve:

1. Factor 10142789312725007 using helper.py or wolfram thus you get $p = 100711409$ $q = 100711423$

2. Obtain decryption exponent $d$ by solving $d = e^{-1} \mod \phi(N) = 8114231289041741$

3. Get the message $m = c^d \mod N = 8283659570766571$
Exercise: Brute Force RSA!

Answer: RSA_FLAG

Steps to solve:

1. Factor 10142789312725007 using helper.py or wolfram thus you get $p = 100711409$ $q = 100711423$

2. Obtain decryption exponent $d$ by solving
   $$d = e^{-1} \mod \phi(N) = 8114231289041741$$

3. Get the message $m = c^d \mod N = 8283659570766571$

4. Break message into ascii aka. $82 = R$, $83 = S$, $65= A$ ...
Review

- Asymmetric cryptography relies on the Discrete Log Problem.


- Want to know more? Take a Cryptography class. CIS 410 this Spring, or MATH 458 Spring ‘16.