Asymmetric Cryptography

And hashing. And other stuff

Zack Orndorff

Problems with Symmetric Cryptography

- Keys have to remain secret
- But you need a secret key in order to safely transmit secrets
- You have a chicken-and-egg problem

- What if there were some way to communicate keys over an insecure channel

Diffie-Hellman key exchange

- Named after Whitfield Diffie and Martin Hellman
  - they also worked with Ralph Merkle, so you'll sometimes see this referred to as Diffie-Hellman-Merkle key exchange
- It's a way for two parties to securely agree on a cryptographic key using insecure communication, without anyone listening able to derive it.
- No one party can choose the key, but that's okay, they just need the same one.
- Uses number theory, resulting in some seemingly basic level math (at least at the conceptual level)
- Let's work it out:

Diffie-Hellman key exchange: *Algorithm*

- Choose two non-secret values p and g, such that p is prime, and g is a primitive root mod p
  - You don't need to know what a primitive root mod n is to understand D-H. It has something to do with finite fields, but let's ignore that for now. There's a reason we tell you not to implement it without more study :)
- Each person chooses a random integer x from {1, …, p-1}.
- They then calculate $Y = g^x$ (mod p). Y can be transmitted in the clear.
- The other person then calculates $K = Y^x$ mod p. (The other person's Y.)
- This leaves both people with the same K, which can be used as a symmetric key.

Link to Diffie and Hellman's original paper:
https://www-ee.stanford.edu/~hellman/publications/24.pdf

Diffie-Hellman key exchange: *Explanation*

- Why is this secure?
- An attacker has g, p, Y_1, Y_2. They need to calculate $K = Y_2^{x1}$(mod p).
- So they need x_1 (or x_2). They can get that with $\log_g Y_1$ (mod p).
- The cryptography community thinks that’s hard to calculate, and that idea has held up for decades.
- There is no mathematical proof that a log mod p is actually hard to solve. This is called the “discrete logarithm” problem, and a fast solution would make some waves in the security field.
- Interestingly, the field is starting to move to elliptic curve Diffie-Hellman, which is believed to be even harder to do.

RSA

- RSA: Stands for Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman, the three inventors of RSA.
- Is not necessarily a method for key exchange (although you could use it that way)
- Uses two keys: the **public key** and the **private key**
- Public key can be published to the entire world
- Private key must be kept secret.

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Algorithm*

- There will be a substantial amount of handwaving in this explanation to try to emphasize the points that will be helpful to understand.
- Key generation:
  - Generate 2 **random prime numbers**, normally denoted **p** and **q**. They must be kept secret.
  - Calculate **n** = p * q. n will be part of your public key.
  - Choose a **public exponent e**. Today's software uses **e=65537** (0x10001) to make math easier.
  - Calculate the **private exponent d** so that e is congruent to d mod (p-1) * (q-1)
    - Here's a substantial amount of handwaving.
  - Your **public key is n and e** (remember, basically everyone uses e=65537
  - Your **private key is n and d** (you generally save p and q as well)

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Algorithm*

- Encryption:
  - Convert your message to a number between 0 and n-1, we'll call it m.
    - There are intricacies here that are critical for security. We're skipping them for clarity.
  - Ciphertext $C = m^e \pmod{n}$ [Remember e and n were our public key]
- Decryption
  - Plaintext $m = C^d \pmod{n}$ [Remember d and n were our private key]
  - Convert m back into your message
- Proof
  - The proof is based on number theory, and requires advanced enough properties that we won't prove it in class.
  - You're welcome to read the paper, though, the proof is in section 6 (VI).

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Explanation*

- Why is this secure?
- You're probably wondering why you can't just factor n into p and q, and calculate the private key.
- We think factoring large products of large primes is hard. Again, there's no proof of this. It's getting easier with modern computers, that's why we use longer and longer keys.

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Uses*

- Encryption (obviously)
- Signing
    - Interestingly, the encryption and decryption operations are inverses of each other, so if something is encrypted with the private (*normally decryption*) key, it can be decrypted with the public key.
    - This has the interesting property that since only the person holding the private key can encrypt with it, it follows that if something can be decrypted with the public key, it was encrypted by the person holding the private key.
    - This is a method of accomplishing **digital signatures**. Typically you take a hash of a message you want to sign and encrypt it with the private key. To verify, you decrypt the signature using the public key and see if the hash matches what it should be.

RSA: *Demo*

- Let’s look at a GPG key.

zack@sperfari: ~


```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFbx/tkBEADAFEI3KZx88Ky/75MNDYYF81QqLc+tLhdsyNBIiAiUDFCbwuOj
+I7Zd3Ix5DzS73kEcbBBqSbnbRnEDETORaL6TPeX7kvrienJfojXpEpjCAa/KGEB
naZ5OkabfbvjiLW/0I0UY7D5xPnc13euT4CaiZ0tppf5G28mbiJKjQQZfMBgqSH4
XDU7M42aKFzRMtrvaV6W/U06GjKGGY6mM1ALa/3vyXTn3jbfosvrz8GHooYEGpue
KgV54N8Tbipt3UW6VhfpUpf/2c277dQiAoklsObhiv3ggRK4f1Cxr87oLIy3pCUR
uYSlnEZz/6i9hHdEkqPIMCCOmtJCiwPf9bZ9M6f0+b2nJ9M4O3ihuxYASAUYPwlW
1CTTF320KQZbwNVQWUFNSHNL2uyxietRq/7G3lIIJTeZD2bfMi9r5AeYpNfdqq/z
QHfVu4fP7J5HFSHx7Pn/xscKVqvairKVVS73DvdSkCMA2/5hTa+PGijYbCT5vgFG
G7AD/BdxceDN4ga9fQG+KHhUfiIFXNT9vYZlgw+4yDD3BEJrxr60JLOeItwBQ0Po
5P1yQ0CpzxozpKq0jSF72gS2JgGzIiC2k6tUodUx4IT1UmYkfKU04N+4KyvK1O1F
11LuAQjat5uqFhurt4RuGkXueYJqtBTRV4g2Z7sqryXjizfZRgjlun1tMQARAQAB
tChaYWNoYXJ5IE9ybmRvcmZmIDx6YWNrQHphY2tvcm5kb3JmZi5jb20+iQI9BBMB
CAAnBQJW8f7ZAhsDBQkDwmcABQsJCAcDBRUKCQgLBRYDAgEAAh4BAheAAAoJEDI4
QOn8Ma+qfIUP/i5UZzTcvmS56pUUK9GyYrkQYJdrhn4aVmxgIS2MYWDW+VHA1scm
jWqK/qOuboFdDCRpTc0iwZ5By3X8DRQZMUbav/PPemN6daQe+rYaEuQP9HfwCOZe
0PQC8iumgkGk/N/UZBiOv/wsXx1VoqeO7hoRXGQ86BTkmbKwWn8Xuor/V4WnT/g7
Vf0vNzthxamXCjxuKWjHQ4nLCdXCCUtfT0qUSoxmcivlX+PbdpNzEu912cHCWM+r
DUc8hNG3/x9Ip/daPEYINakDuc21Sd+wQ52r07y6yQuacz2c5qCGup1ROalt/GNq
GqmFcpwhoyOM88fwC15hY81IrDd2dTFVFZ6TOvihwc4qb/ENmAmuJfxasFQ4wcGW
GCkSJxCwjjlg8Z5Uo6r7lMj5P6MPTbsqrhMhmzxr1xyYqnmZdsdHKRrkGKXzSqIw
L1JGkIVFahctSBFjy/hZWLb9DJAYrK6iX99OFU0vtXc94r4iM4Yui/MDRrpn2byX
1GT2YjTVrQgcTW1UUD6bTOSH6D6BNZ5hXsU6l0nAclDtYPaafQQPUcTL/SCkyL4y
--More--(27%)
```


zack@sperfari: ~

```
$ pgpdump -i Documents/zack.public.gpg-key
Old: Public Key Packet(tag 6)(525 bytes)
        Ver 4 - new
        Public key creation time - Tue Mar 22 22:26:33 EDT 2016
        Pub alg - RSA Encrypt or Sign(pub 1)
        RSA n(4096 bits) - c0 14 42 37 29 9c 7c f0 ac bf ef 93 0d 0d 86 05 f3 54 2a 2d
 cf ad 2e 17 6c c8 d0 48 88 08 94 0c 50 9b c2 e3 a3 f8 8e d9 77 72 31 e4 3c d2 ef 79 0
4 71 b0 41 a9 26 e7 6d 19 c4 0c 44 ce 45 a2 fa 4c f7 97 ee 4b eb 89 e9 c9 7e 88 d7 a4
4a 63 08 06 bf 28 61 01 9d a6 79 3a 46 9b 7d bb e3 88 b5 bf d0 8d 14 63 b0 f9 c4 f9 dc
 d7 77 ae 4f 80 9a 89 9d 2d a6 97 f9 1b 6f 26 6e 22 4a 8d 04 19 7c c0 60 a9 21 f8 5c 3
5 3b 33 8d 9a 28 5c d1 32 da ef 69 5e 96 fd 4d 3a 1a 32 86 19 8e a6 33 50 0b 6b fd ef
c9 74 e7 de 36 df a2 cb eb cf c1 87 a2 86 04 1a 9b 9e 2a 05 79 e0 df 13 6e 2a 6d dd 45
 ba 56 17 e9 52 97 ff d9 cd bb ed d4 22 02 89 25 b0 e6 e1 8a fd e0 81 12 b8 7f 50 b1 a
f ce e8 2c 8c b7 a4 25 11 b9 84 a5 9c 46 73 ff a8 bd 84 77 44 92 a3 c8 30 20 8e 9a d2
42 8b 03 df f5 b6 7d 33 a7 f4 f9 bd a7 27 d3 38 3b 78 a1 bb 16 00 48 05 18 3f 09 56 d4
 24 d3 17 7d b4 29 06 5b c0 d5 50 59 41 4d 48 73 4b da ec b1 89 eb 51 ab fe c6 de 52 0
8 25 37 99 0f 66 df 32 2f 6b e4 07 98 a4 d7 dd aa af f3 40 77 d5 bb 87 cf ec 9e 47 15
21 f1 ec f9 ff c6 c7 0a 56 ab da 8a b2 95 55 2e f7 0e f7 52 90 23 00 db fe 61 4d af 8f
 1a 28 d8 6c 24 f9 be 01 46 1b b0 03 fc 17 71 71 e0 cd e2 06 bd 7d 01 be 28 78 54 7e 2
2 05 5c d4 fd bd 86 65 83 0f b8 c8 30 f7 04 42 6b c6 be b4 24 b3 9e 22 dc 01 43 43 e8
e4 fd 72 43 40 a9 cf 1a 33 a4 aa b4 8d 21 7b da 04 b6 26 01 b3 22 20 b6 93 ab 54 a1 d5
 31 e0 84 f5 52 66 24 7c a5 34 e0 df b8 2b 2b ca d4 ed 45 d7 52 ee 01 08 da b7 9b aa 1
6 1b ab b7 84 6e 1a 45 ee 79 82 6a b4 14 d1 57 88 36 67 bb 2a af 25 e3 8b 37 d9 46 08
e5 ba 7d 6d 31
        RSA e(17 bits) - 01 00 01
```

Cryptographic Hashes

- Names like “MD5” “SHA-1” “SHA-256” “SHA-3” “Keccak”
- Different from hash functions for hash tables
- They’re *one-way functions*
- Properties:
    - Preimage resistance
        - Given hash, find a message
    - Second preimage resistance
        - Given message, find a message with same hash
    - Collision resistance
        - Find 2 messages with same hash

```
$ md5sum /bin/ls
0b19809bab331d70fb9983a0b9866290  /bin/ls
```

Cryptographic Hashes: *Fun Facts*

- MD5 - 128 bits
  - Looks like this: 0b19809bab331d70fb9983a0b9866290
  - Considered completely broken
- SHA-1 - 160 bits
  - Looks like this: 85c05d8c0e085040e5eda54de4638d4485b1d22d
  - Considered broken, because Google threw a Google-sized amount of power at it and found a collision
- SHA-256 - 256 bits
  - Looks like this: a0e06b5a72fed6c106391cf0162dbee1750c047640117bfea2234857055216a0
- SHA-3 - size varies, just like SHA-2
  - Uses “Sponge” construction. All the rest use the Merkle-Damgard construction

```
$ md5sum /bin/ls
0b19809bab331d70fb9983a0b9866290  /bin/ls
```

MACs: Message Authentication Codes

- These prove the integrity of a message.
- They’re kind of like a hash (and can be based on hash functions)
- Names you’ll hear are:
  - HMAC
  - Poly1305
- Process:
  - 1. Generate/distribute a symmetric key
  - 2. “sign” the message (append a tag to it that ensures integrity)
  - 3. Verify the “signature”/tag
    - I use “signature” in quotes since it’s a symmetric tag, not an asymmetric signature

Bits about Password Hashing

- Cryptographic hashes are designed to be fast, to handle lots of data
- Password hashes are designed to be slow, to slow down the attacker
- Don’t MD5 passwords. Don’t SHA-256 them either
- Use something like bcrypt or PBKDF2 (or Argon2)
- Bcrypt hashes look like:
  $2b$12$2mvGuZugvDdi2Q/T./e3L.RO.Ht9updiwhGk8ntVGf5AsJsQ2S10O

Random Number Generation

- rand() is not an acceptable random number generator for… well really anything
- If you want something statistically good, go find an actually good pseudo-random number generator (PRNG), libc’s is junk.
- If you’re going to use the numbers for security-related purposes, use a **cryptographically secure random number generator (CSPRNG)**.
  - If you don’t know that it’s a CSPRNG, it probably isn’t.

Quantum Computing and Cryptography

- If a **sufficiently large** quantum computer is ever built:
  - RSA and Diffie-Hellman are **completely broken** by an algorithm called Shor's algorithm
  - The **bit length** of symmetric ciphers is **effectively halved**. I.E. if it would previously required 2^{128} computations to crack something, it would require 2^{64} quantum computations.
  - Hash functions - in general preimage resistance is halved, and collision resistance is decreased from $2^{n/2}$ computations to $2^{n/3}$ quantum computations.
- More info: post by cryptographer Thomas Pornin on Stack Exchange:
  - http://security.stackexchange.com/questions/48022/what-kinds-of-encryption-are-not-breakable-via-quantum-computers/48027#48027
- Cryptographers are working on encryption algorithms that are not as vulnerable to quantum cryptography.

Resources to learn more

- About the history of cryptography: *The Code Book* by Simon Singh
- The Cryptopals Crypto Challenges: https://cryptopals.com/
  - Formerly the Matasano Crypto Challenges
- The crypto course here at UMBC