

CITS3007 Secure Coding

Introduction to cryptography

Unit coordinator: Arran Stewart

Highlights

- ▶ Ciphers
- ▶ Symmetric-key and public-key cryptography
- ▶ Cryptographic hash functions
- ▶ salt

Cryptography

Cryptography: the study of techniques for secure communication in the presence of third parties

- ▶ In other words, applicable to any situation where we want to make sure a given message can be read by only the sender and receiver

Cryptography

Applications of cryptography:

- ▶ Communicating securely with websites
 - ▶ We don't want others to be able to read our requests and passwords
- ▶ Transferring funds
- ▶ Storing user information in a database
 - ▶ e.g. credit card details, passwords
 - ▶ “receiver” of a message might just be ourselves, but at a later time

Terminology

plaintext a message we want to encrypt

ciphertext the encrypted message

encryption the process of encoding a message such that only the authorized parties can access it

decryption the process of decoding a given message

key a sequence that needed both encryption and decryption

Simple example – Caesar cipher

A **cipher** is a pair of algorithms that encrypt (convert plaintext to ciphertext) and decrypt (convert ciphertext to plaintext).

A very simple example is the *Caesar cipher*:

- ▶ Assume for simplicity our message consists only of letters from the English alphabet.
- ▶ We have a *key*, which is some number from 1 to 26.
- ▶ To *encrypt*, we shift every letter “along” by *key* many places
 - ▶ e.g. If our key is 3, then ‘A’ becomes ‘D’, ‘B’ becomes ‘E’, ‘Z’ becomes ‘C’, etc.
- ▶ To *decrypt*, we just shift back
 - ▶ e.g. ‘D’ becomes ‘A’, ‘E’ becomes ‘B’, ‘C’ becomes ‘Z’, etc.

Simple example – Caesar cipher

The Caesar cipher is an example of a *substitution cipher* – each letter in the original message is substituted with some other letter.

- ▶ If we know something uses a simple substitution cipher like this, then it's very easy to attack (especially if we have plenty of ciphertext)
- ▶ Just measure the letter frequency of the ciphertext – the most common letter is most likely 'E', the next most common probably 'A', and use a little guesswork to find out the key
 - ▶ The most common letters in English are found in the nonsense words “**ETAOIN SHRDLU**”

Ciphers versus codes

So that's a cipher.

A **code**, on the other hand, is just a way of mapping one representation into another.

For example

- ▶ ASCII maps English letters and numbers (plus punctuation and some other special symbols) into a 7-bit number
- ▶ Morse code maps English letters and numbers into sequences of dots and dashes

Types of cryptography

Two basic types of encryption method:

- ▶ symmetric-key cryptographys
 - ▶ both parties use a single (shared) key for encryption and decryption
- ▶ public-key cryptography
 - ▶ each party has two keys – a *public key* (which other people know) and a *private key* (which they don't)

In addition to these, we also make use of *cryptographic hash functions*.

Symmetric-key cryptography

Symmetric-key cryptography uses a single key for encryption and decryption.

- ▶ The Caesar cipher is an example of this.
- ▶ They use a “shared secret” (the key) known by the sender and receiver
- ▶ Up until 1976, when public-key cryptography was invented, this was the only known form of cryptography

Example: **AES** (Advanced Encryption Standard) cipher

- ▶ Developed in 1990s to replace a previous standard, DES
- ▶ Uses keys of length 128, 192 or 256 bits
- ▶ 128-bit AES currently considered safe against brute-force attacks

Public-key cryptography

Basic idea hit on in 1874 by William Stanley Jevons:¹

“Can the reader say what two numbers multiplied together will produce the number 8616460799? I think it unlikely that anyone but myself will ever know.”

Multiplication is easy, but factorisation is hard.

The above example can be solved quickly using computers, but would've been very difficult in 1874.

By increasing the size, we can come up with numbers which are still easy for computers to multiply, but difficult to factorize.

Factorizing a 240-digit (795-bit) number will take around 900 core-years of computing time.

¹In *The Principles of Science*

Public-key cryptography

Example of public-key cryptography: RSA (Rivest–Shamir–Adleman) cryptosystem.

- ▶ Used by e.g. `ssh` – you use `ssh-keygen` to create public and private keys stored in `~/.ssh` directory
- ▶ `id_rsa.pub`: public key, you can give this to anyone
- ▶ `id_rsa`: private key, you keep this secret

Public-key cryptography

Suppose Alice wants to send a message to Bob.

- ▶ She can encrypt a message using Bob's *public* key.
- ▶ Only Bob can decrypt such a message, using his *private* key.

Suppose Bob wants to be able to easily prove who he is to (say) GitHub.

- ▶ He provides GitHub with his SSH *public* key.
- ▶ Later on he wants to authenticate. GitHub encrypts some random text with Bob's public key, and sends the encrypted text to Bob.
- ▶ Only Bob can decrypt the message – he does so, and sends GitHub back the random text they encrypted, proving that it's him.

Hash functions

A **hash function** is some function that operates on arbitrary data (so we may think of it as a list of bytes) and maps it to some fixed-size value (usually a number).

So we may think of it as:

```
hash(value: array<byte>) → vector<byte, N>
```

for some fixed N .

Example: The MD5 algorithm is a hash algorithm.

- ▶ It takes *in* any arbitrary list of bytes, and outputs a 128-bit number.

Cryptographic hash functions

Good **cryptographic hash functions** have the following properties.

- ▶ Deterministic: the same input always generates the same output
(though this follows from it being a function)
- ▶ Fast to compute
- ▶ Difficult to invert: it is hard to find an input m to the function such that $hash(m) = h$ for some desired output hash h
- ▶ Resistant to target collision: given some input m_1 , it is difficult to find another input m_2 that hashes to the same output (i.e. such that $hash(m_1) = hash(m_2)$)

NB that MD5 is no longer considered a good cryptographic hash function, because modern computers are powerful enough to find collisions.

Nor is **SHA-1** (since 2005), but SHA-2 and SHA-3 are.

Cryptographic hash function use

When “storing” user’s passwords, *don’t* actually store plaintext password

- ▶ Very bad if we are compromised
- ▶ Instead store a *hash* of the password
 - ▶ this proves (to whatever degree of certainty we would like) that someone *knows* the password, without us having to store it

Hash functions in `/etc/shadow`

Recall from labs that in `/etc/passwd`, often the “password” field is just an “x”, meaning that a hash is stored in `/etc/shadow`

A record in `/etc/shadow` looks like:

```
bob:$6$3fCW76UTZApV/cMu$0gP... (omitted) ... NKmq/:18949:0:99999:7:::
```

Hash functions in `/etc/shadow`

```
bob:$6$3fCW76UTZApV/cMu$0gP... (omitted) ... NKmq/:18949:0:99999:7:::
```

8 fields separated by colons:

1. Username
2. Hashed password
3. Last password change
4. Minimum password age
5. Maximum password age
6. Warning period
7. Inactivity period
8. Expiration date

Some fields may be empty if the system doesn't use them
(e.g. expiration date)

Hash functions in `/etc/shadow`

```
bob:$6$3fCW76UTZApV/cMu$0gP... (omitted) ... NKmq/:18949:0:99999:7:::
```

- ▶ The “hashed password” field actually has 3 subfields – it’s format is `$type$salt$hashed`

Where “type” is:

- ▶ `1` – MD5
- ▶ `$2a$` – Blowfish
- ▶ `$2y$` – Eksblowfish
- ▶ `5` – SHA-256
- ▶ `6` – SHA-512

Hash functions in `/etc/shadow`

```
bob:$6$3fCW76UTZApV/cMu$0gP... (omitted) ... NKmq/:18949:0:99999:7:::
```

The “salt” is just some random value. Rather than hash the password directly, we hash password + salt (concatenated)

Why salt?

Suppose we have access to a bunch of hashed passwords.

We know what hash algorithm was used to create them (e.g. SHA-512).

So we could just hash the most common passwords people use (“password”, “abc”, “qwerty”, “123456”) and see if those hashes turn up in `/etc/shadow`.

If they do – voilà, we’ve cracked their password.

But if instead what was hashed is (password + salt), this attack no longer works – even if we know what the salt was.

Why salt?

A similar modern technique used by attackers are *rainbow tables*:

- ▶ rainbow table: an efficient way to store data that has been computed in advance to facilitate cracking passwords

Salting prevents the use of rainbow tables.

recommendation

Never store passwords in plaintext. Store a (salt, hash) pair, where the “hash” is the hashed password + salt.