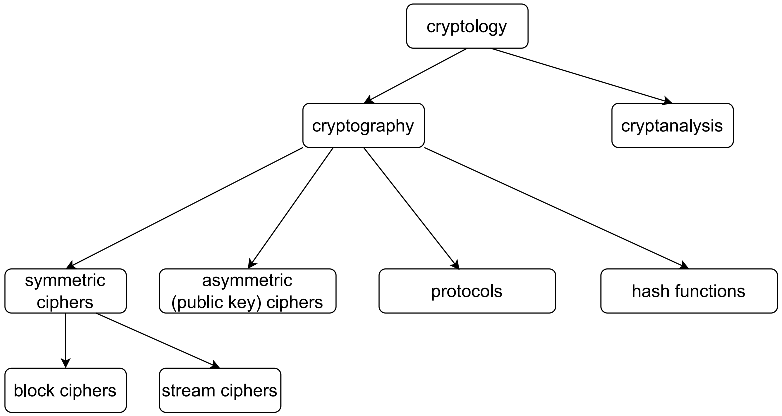
CITS3007 Secure Coding
Introduction to cryptography

Unit coordinator: Arran Stewart

Highlights

- Ciphers and codes
- Symmetric-key and public-key cryptography
- Crypto building blocks
- Cryptographic hash functions
- Password storage

Overview of field



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

Cryptanalysis: attempting to find weaknesses in cryptographic routines.

- Why do we need it?
- Because currently, the only way we have knowing whether a new cryptographic techniques "works" is by trying to break it and failing.
- There's only one cryptographic technique that's *provably* unbreakable – the one-time pad (Shannon 1949) – which for pragmatic reasons is not much used

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
- Validating that content hasn't been tampered with (cryptographic signing)

Applying cryptography

Cryptography is obviously immensely useful in helping to achieve security goals:

- confidentiality
- integrity
- authenticity

However:

- Cryptography on its own won't achieve those goals – it has to be applied appropriately
- Cryptography is **easy to get wrong** – you can introduce major security vulnerabilities if you don't know what you're doing
- Cryptography is [for most of us] **not** something you should ever implement yourself
  - Considerable expertise and validation of designs is required when implementing new cryptographic libraries or technologies

Cryptography pitfalls

We said in lecture 2 that API quality can range from excellent (+10, “Impossible to use incorrectly”) to appalling (-10, “Impossible to use correctly”).

Some popular cryptography libraries have fairly low quality APIs, measured using this rubric – they are somewhere below level 3 (“Read the documentation and you’ll get it right”).

You have to read the documentation *very* carefully in order not to make catastrophic mistakes.

Cryptography pitfalls – API misuse

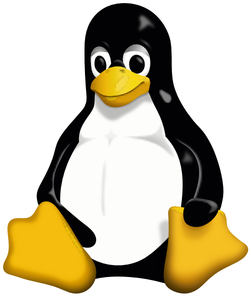
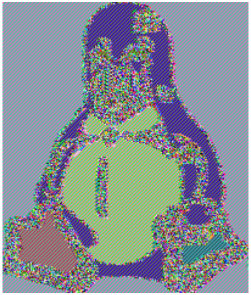
Example: ECB mode

- One cipher we look at is AES (used e.g. in SSH).
- It's what's called a **block cipher** – it operates on data in fixed-size blocks.
    - If you're using "128-bit AES", then the data is split up into 128-bit (16-byte) sized blocks.
- You then have to specify a **block mode**: how to apply the cipher when you have more than a single block's worth of data (usually the case). (More on this later.)
- If you happen to select a mode called "ECB" ("Electronic Code Book"), then you'll make your encryption easily crackable.

Cryptography pitfalls – ECB penguin

Example: ECB mode

Using ECB mode makes any patterns in the original data very visible in the encrypted data.

The "ECB penguin". Original data (left) and data encrypted using ECB (right). Credit: user Lunkwill of Wikipedia, 2004.

Cryptography pitfalls – API misuse

Example: WinCrypt.h

- We know it's a bad idea to "roll your own" cryptography routines
- So if you are on Windows, it makes sense to use the cryptography APIs provided by Windows – one of these is "WinCrypt.h"
- It has multiple "providers" (e.g. the "Microsoft Diffie-Hellman Cryptographic Provider"), you need to choose one and get a "handle" to it using `CryptAcquireContext()`
- The API documentation had an example of use of this function

```
CryptAcquireContext(
  &hCryptProv,   // handle to the CSP
  UserName,      // container name
  NULL,          // use the default provider
  PROV_RSA_FULL, // provider type
  0);            // flag values
```

Cryptography pitfalls – API misuse

Example: WinCrypt.h

```
CryptAcquireContext(
  &hCryptProv,    // handle to the CSP
  UserName,       // container name
  NULL,           // use the default provider
  PROV_RSA_FULL, // provider type
  0);             // flag values
```

- But if you use the provided code – passing 0 for the “flag values” means that the private key is kept in the local “key-store” on Windows.
- Ransomware writers called the function in this way to encrypt victims’ data
- But since the private key needed to decrypt the data was still in the key-store of victims’ machines, the data was easily recoverable.[1]

[1] Emsisoft (2014), “CryptoDefense: The story of insecure ransomware keys and self-serving bloggers”

Cryptography pitfalls – SaltStack

Example: Saltstack encryption keys

- SaltStack is a tool (now owned by VMWare) used for configuring and managing large numbers of servers and tasks.
- It used the RSA cryptosystem to ecnrypt messages sent between servers
- A SaltStack developer wrote the following code to create an RSA public key using the pycrypto library:

```
gen = RSA.gen_key(keysize, 1, callback=lambda x, y, z: None)
```

Cryptography pitfalls – SaltStack

Example: Saltstack encryption keys

```
gen = RSA.gen_key(keysize, 1, callback=lambda x, y, z: None)
```

- The second parameter (1) is what's called the "public exponent" for the cryptosystem – it's one of a pair of 2 numbers that make up the public key.
- Unfortunately, 1 is a terrible choice – it makes the cryptography easy to crack.[2]
- SaltStack had to inform users that their encryption keys had been generated insecurely, and that they should re-generate all keys.[3]

[2] StackOverflow (2014), "Why is this commit that sets the RSA public exponent to 1 problematic?".

[3] Salt Project (2013), "Salt 0.15.1 Release Notes".

Cryptography pitfalls – don't "roll your own"

Example: IOTA DIY hash function

- In 2017, the cryptocurrency IOTA was the 8th largest cryptocurrency (with $1.9 billion market capitalization).
- It made use of cryptographic *hash functions*
- Rather than use existing hash functions that were known to work, the developers decided to implement their own, called "Curl"
- Cryptographers analysed the algorithm and found critical weaknesses in it[4]

[4]Neha Narula (2017), "Cryptographic vulnerabilities in IOTA"

Cryptography pitfalls – IOTA

Example: IOTA DIY hash function

- To maintain viability of the cryptocurrency, IOTA developers were forced to switch to a standard hashing algorithm, SHA3
- Per one of the cryptographers, Neha Narula:

  . . . [W]hen we noticed that the IOTA developers had written their own hash function, it was a huge red flag. It should probably have been a huge red flag for anyone involved with IOTA.

Basics

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 *monoalphabetic substitution cipher* – each letter in the original message is substituted with some other letter.

- If we know a message uses a simple substitution cipher like this, and we know it's written in English, then the cipher is 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 punctutation 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 cryptography

- also called “shared key” cryptography
- a single key is used, which both encrypts and decrypts
- example: Caesar cipher, the key is an integer to shift by
- example: an encrypted “zip” file – a key is specified when the file is created.

public-key cryptography

- each party has two keys – a *public key* (which other people know) and a *private key* (which they don’t)
- example: an SSH key pair – you can “prove that you are you” to servers which hold a copy of your *public* key, because only you have a copy of your *private* key.

In addition to these, we also look at *cryptographic hash functions* (used for authentication).

Symmetric-key cryptography

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

- 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

Symmetric-key example – Caesar cipher

- The Caesar cipher is an example of this. It is a type of cipher known as a "monoalphabetic substitution cipher"
  - (meaning: that for any letter, it's replaced, wherever it appears, by some other letter)
- The "shared key" is just a number (e.g. 3) which represents the number of "places" to shift each letter.
- Monoalphabetic substitution ciphers like these are easily cracked
  - Brute force: there are only 25 possible keys – trivial to try them all
  - Frequency analysis: "e" is the most common letter in English (the most common 12 are "etaoinshrdlu"), so the most common letter in the ciphertext probably represents "e".
  - We can use that plus some guesswork to quickly work out the key.

Symmetric-key example – AES

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

Symmetric-key example – AES

AES is one of the symmetric ciphers used by the SSH protocol.

The client and the server derive a secret, shared key using an agreed method, and the session is encrypted using that key.

The initial connection and negotiation of this shared key uses asymmetric encryption; but symmetric encryption is much faster than asymmetric, so it's used for the remainder of the session.

No-one has *proved* that AES is secure; but it has been thoroughly investigated by many cryptographers, and all attempts to break it have failed.

Public-key cryptography

Also called "asymmetric cryptography" or "asymmetric encryption"

Basic idea hit on in 1874 by William Stanley Jevons:[5]

> *"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.

[5] In *The Principles of Science*

Public-key cryptography

Published public-key cryptographic systems did not appear until the 1970s.

Example of public-key cryptography: RSA (Rivest–Shamir–Adleman) cryptosystem (1977).

- 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.

Cipher building blocks

Most ciphers make use of two basic techniques: **substitution** and **transposition**.

(For simplicity, we'll phrase these concepts in terms of "letters"; but in modern ciphers, we would actually apply them at the bit level.)

substitution	Substitute letters in the plaintext with other letters, according to some rule.
transposition	Scramble/reorder the letters in the plaintext, according to some rule.

May seem simple – but even modern symmetric ciphers use these two techniques.

Substitution

- We've seen how monoalphabetic substitution works – the Caesar cipher does substitution by a simple rule ("Add K to each letter, modulo the size of the alphabet"), and no transposition
- More complex rules might make different substitutions depending on what position we're at in the plaintext (e.g. we might cycle through six rules), or might take e.g. *pairs* of letters and look up a table to see what to substitute

Transposition

- Example of a simple transposition rule: the **rail fence cipher**
- Write your plaintext across e.g. 3 "rails", zig-zagging across them
- Then read off the ciphertext "horizontally"

Rail fence cipher

plaintext `THIS WAS A TRIUMPH.`

rails

T				W				T				M		
	H		S		A		A		R		U		P	
		I				S				I				H

ciphertext `TWTMHSAARUPISIH`

General principles of ciphers

Ciphers have two characteristics which describe how hard it is for an attacker (given a copy of the ciphertext) to get information about the plaintext and key, **confusion** and **diffusion**.

These characteristics allow a cipher to resist simple statistical analysis.

confusion	A cipher provides good confusion if the ciphertext gives the attacker little information about the *key*.
diffusion	A cipher provides good diffusion if the ciphertext gives the attacker little information about the *plaintext*.

Confusion

confusion

Characteristic of a cipher which describes how hard it is for an attacker, when given the ciphertext, to determine the key

- Compare the Caesar cipher: if given the ciphertext, the simple substitution rule makes it *easy* to determine the key.
- So strong ciphers use much more complex substitution rules. E.g.: "Given 6 bits from the message, look up table T; locate a table row using (b_0, b_5), and a table column using the middle 4 bits, and substitute the contents of the table cell you get."
- If a cipher provides good confusion, changing the key even slightly will result in very different ciphertext.

Diffusion

diffusion

Characteristic of a cipher which describes how hard it is for an attacker, when given the ciphertext, to determine the plaintext

- In the Caesar cipher, the relationship between plaintext and ciphertext is quite simple
- Every letter in the plaintext ends up in exactly the same position of the ciphertext – no transposition is done
- So strong ciphers make use of much more complex rules, and "spread" the influence of a single letter in the plaintext throughout a large portion of the ciphertext
- If a cipher provides good diffusion, changing the plaintext even slightly will result in very different ciphertext.

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 abitrary list of bytes, and outputs a 128-bit number.

Types of hash functions

To implement a hash function at all, a procedure should

- be deterministic: the same input always generates the same output
  (this follows from it being a function)
- map inputs of all lengths into some fixed range of outputs.

Anything that meets those criteria qualifies as a hash function (it might not be a good one, though).

Types of hash functions

Some sub-types of hash function:

non-cryptographic hash functions

- Used for e.g. calculating checksums for files, or in data structures like hash tables.
- Faster and simpler than others
- Not designed to resist deliberate attack
- Example: CRC32

Types of hash functions

cryptographic hash functions

- Used for e.g. authentication, digital signatures, message authentication codes
- Designed to be resistant to deliberate attack
- Have several desirable properties: collision resistance, preimage resistance, and second preimage resistance
- Examples: SHA-2, SHA-3
- Non-examples: MD5, SHA-1
  - No longer suitable for cryptographic purposes
  - SHA-1 broken in 2005 by researchers from Google and Centrum Wiskunde & Informatica (CWI) in Netherlands

Types of hash functions

password hash functions

- Technically called key derivation functions (KDFs) – but password hashing is where you'll most likely encounter them
- Designed to resist brute-force attacks (by e.g. clusters of computers, or dedicated hardware like FPGAs)
- In general, all the other types of hash function we've seen should be fast to run (i.e. have good throughput rate)
- To resist attack, password hash functions are deliberately *slow*
- Examples: Argon2id, scrypt, bcrypt
- Non-examples: everything else. Do *not* use MD5, SHA-1

For advice on choosing an algorithm, see the OWASP Password Storage Cheat Sheet

Cryptographic hash functions

In case it's of interest – the properties of good cryptographic hash functions we mentioned are as follows:

collision resistance
: Should be infeasible to find any two inputs m_1 and m_2 that hash to same output
i.e. such that $hash(m_1) = hash(m_2)$

preimage resistance
: Given a hash, should be infeasible to find a corresponding input that produces that hash
i.e. we can't reverse the function

second preimage resistance
: If we're given an input m_1, should be infeasible to find a second input m_2 that produces the same hash value
(not the same as collision resistance, because here you're given a specific input to match)

Password storage

How to store passwords

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.

References

- Shannon, “A Mathematical Theory of Cryptography”, Bell System Technical Memo MM 45-110-02, 1945 (PDF)
- Shannon, “Communication Theory of Secrecy Systems”, Bell System Technical Journal, vol. 28(4), page 656–715, 1949