COMS 3000/7003

Week 10

Symmetric Cipher Modes,

Asymmetric Cryptography
(Public Key Cryptography)

This week's news

- Man in the middle – via email
- See whiteboard

This week's news

NEWS

Ousted Equifax CEO Faces 3 Congressional Hearings

by Mathew J. Schwartz

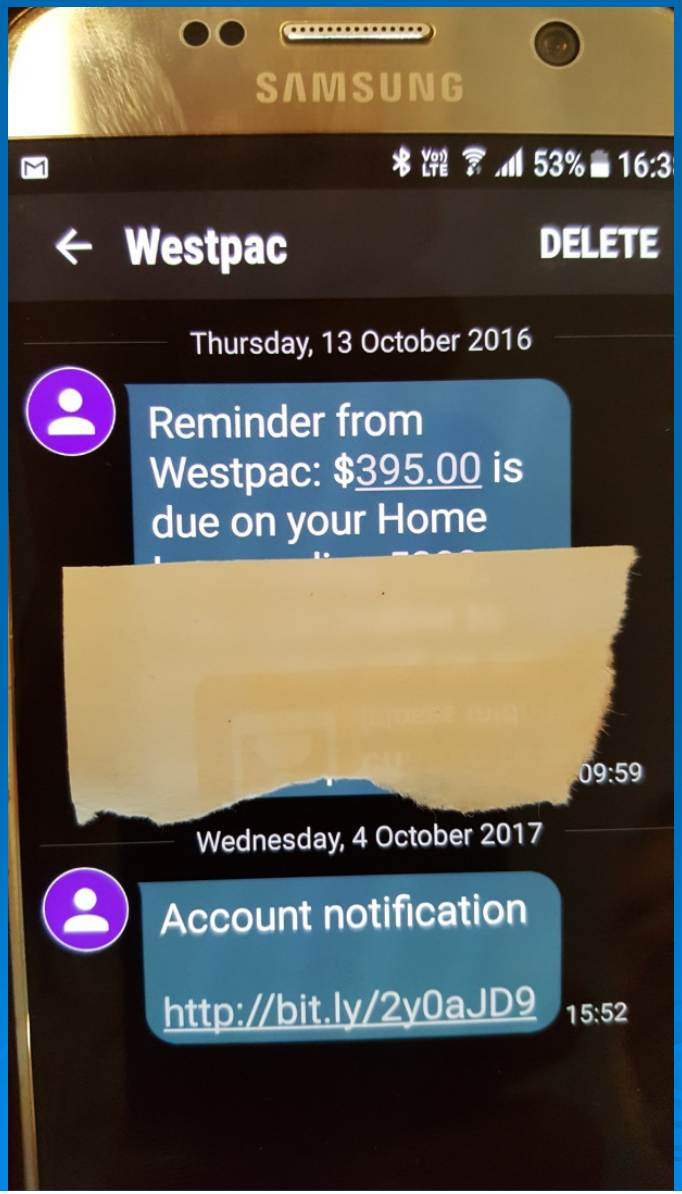
Former Equifax CEO Richard Smith this week heads to Capitol Hill to testify about the massive breach suffered by the credit bureau. Lawmakers will likely focus on breach detection and response, information security practices and the suspicious timing of three executives' stock sales.

This week's news

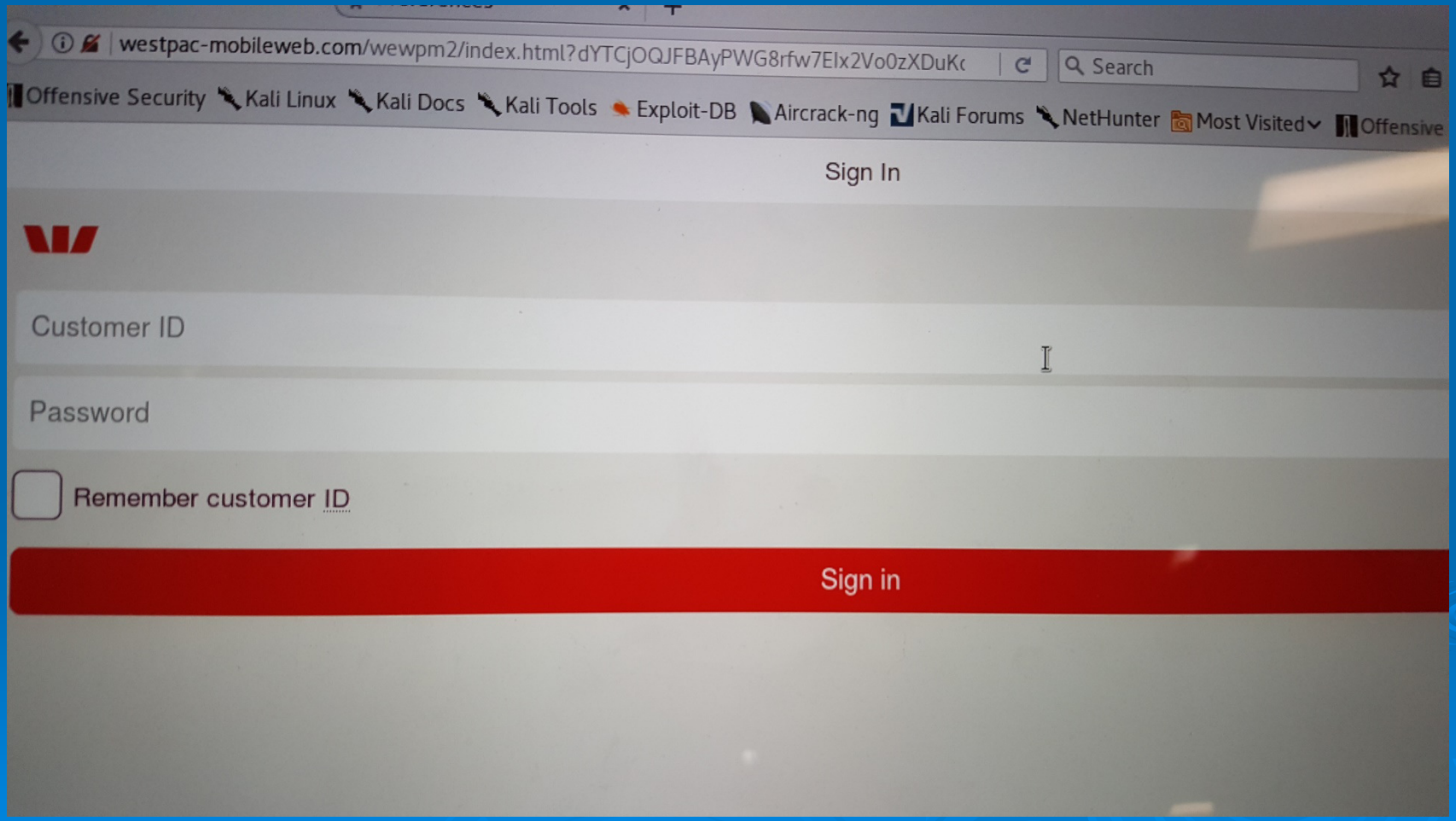
AISA National Conference 10–12 October 2017

11:55 AM - 12:40 PM	Dr Maria Milosavljevic - Building the NSW Government Cyber Security capability	IoT - The internet of Insecure, downright Scary Things...or not? An interactive session chaired by Lani Refiti	Should industry be allowed to engage in 'active defence'?		
12:40 PM - 1:40 PM	**Lunch - Sponsored by Proofpoint**				
1:40 PM - 2:30 PM	Keynote Address with Christopher Painter International diplomacy to counter cyber threats				
2:30 PM - 3:15 PM	Keynote Address with Michael Daniel Changing the game: how altering our mindset, adopting new frameworks, and sharing information among the right players can make us all more secure				
3:15 PM - 3:45 PM	**Afternoon Tea - Sponsored by Proofpoint**				
3:45 PM - 4:45 PM	Keynote Address with Dr Charlie Miller & Chris Valasek				
4:45 PM - 5:00 PM	Conference Closing				

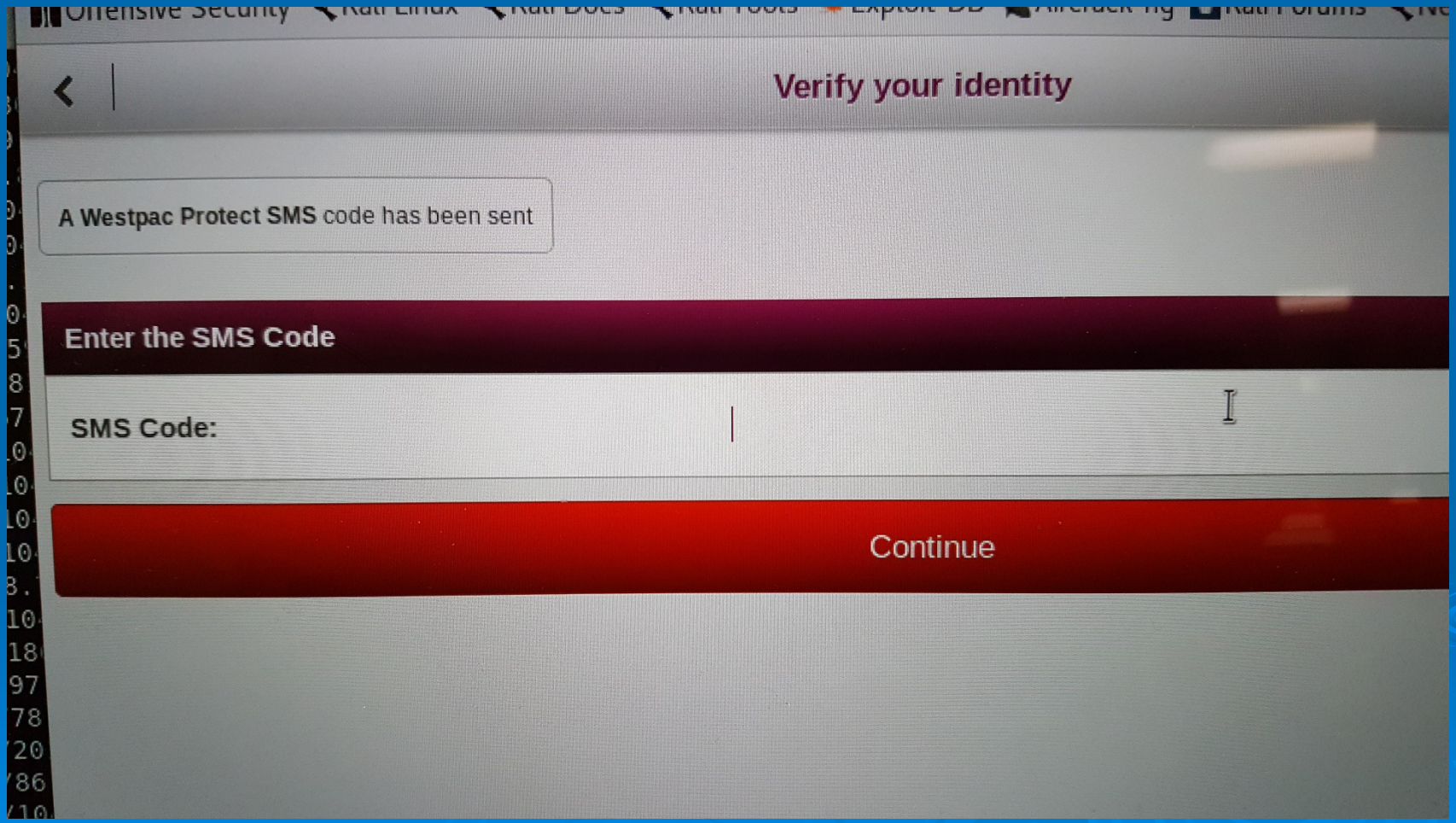
This week's news



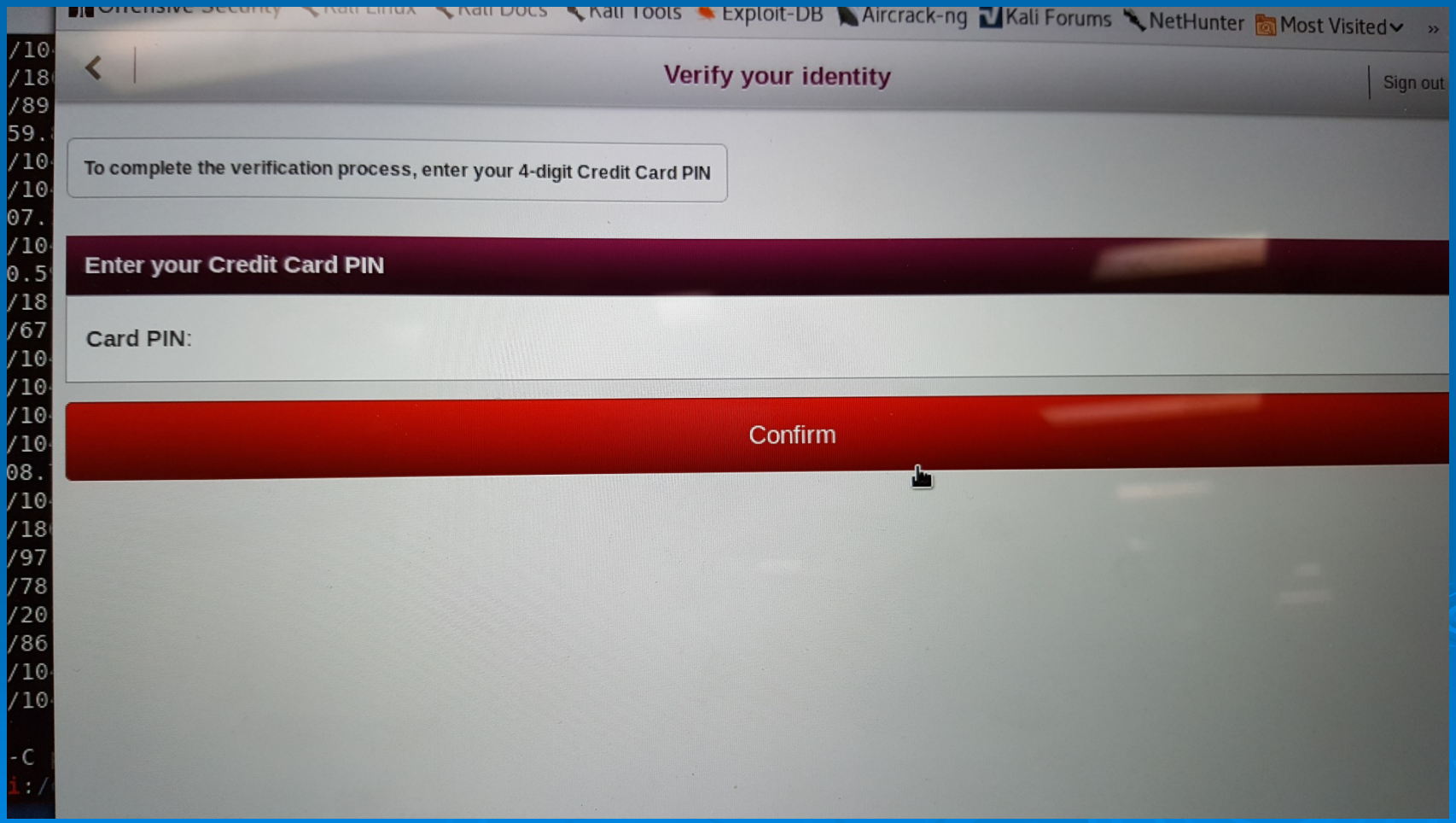
This week's news



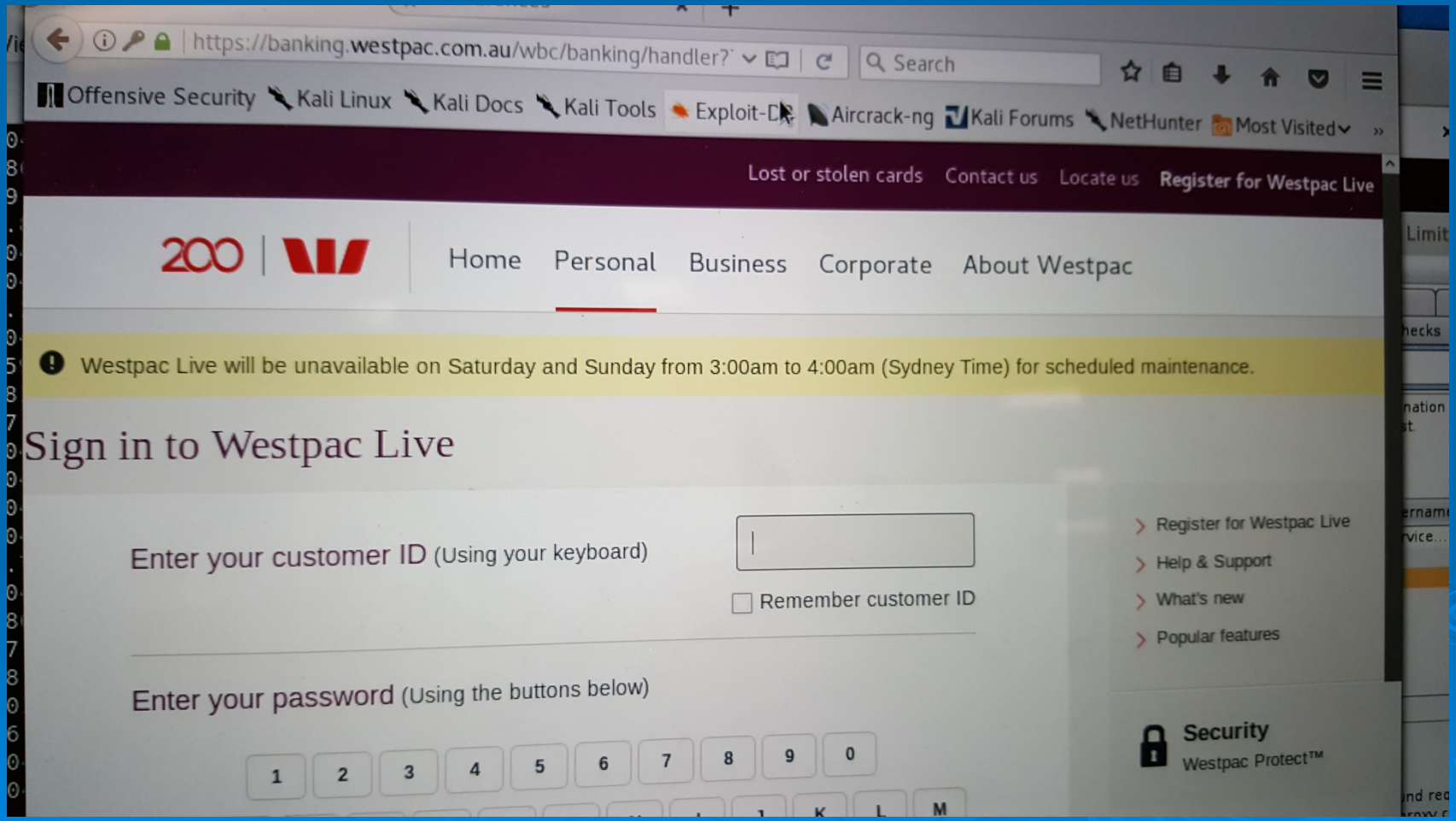
This week's news



This week's news



This week's news



Assignments

- Results are on Blackboard
- Except for extensions and non-submission

Tutor Assessments

Dear students,

UQ and ITEE takes pride in the quality of its teaching staff and encourages you to provide constructive feedback of the course tutors through the SETutor survey process. Before Saturday 22 October, 2017 please access the link below and using the password for the tutors you have had contact with this semester complete the online survey.

Tutor Assessments

Link: https://eval.uq.edu.au/	
Tutor:	**Password:**
Mr Kristan Edwards	TT2ES
Dr Kaleb Leemaqz	5ZPFQ

Thank you, your cooperation is appreciated.

Back to Passwords
Unix Password Process

- Back in Lecture 2 we discussed pre-Multix (-> pre-UNIX -> pre-Linux) password files stored in plaintext…

- Then early UNIX /etc/passwd hashes…
  - Asside: Unix is not an acronym (UNICS was) but was originally published in small caps by Thompson and Ritchie, so:
    UNIX is Unix, trademarked UNIX (The Open Group)

The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11
CR Categories: 4.30, 4.32

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and /45 [1] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Fortran compiler, Snobol interpreter, top-down compiler-compiler (TMG) [4], bottom-up compiler-compiler (YACC), form letter generator,

Attacks against Password Storage

- Problem:
  - Passwords need to be stored somewhere, needed by the system during login process
    - → vulnerability
- Originally, computer passwords were stored in plaintext, i.e. unencrypted
- Illustrative Example of the problem:
  - "In MIT's Compatible Time Sharing System, ctss (a predecessor of Multics), it once happened that one person was editing the message of the day, while another was editing the password file. Due to a software bug, the two editor temporary files got swapped, with the result that everyone who logged on was greeted with the password file."

    R. Anderson, Security Engineering

Attacks against Password Storage

- That was from Robert Morris
  - Robert H Morris Sr cryptographer at Bell Labs
  - Not Prof Robert T Morris creator Morris Worm
- Homework reading (very short 4 pages):
  - Password Security: A Case History, Robert Morris and Ken Thompson, Bell Laboratories
  - There is a good copy on Peter Guttman's site: http://web.cs.wpi.edu/~guttman/cs557_website/papers/passwords/MorrisThompsonPasswordSecurity.pdf

Operating Systems
R. Stockton Gaines Editor

Password Security: A Case History

Robert Morris and Ken Thompson
Bell Laboratories

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

Key Words and Phrases: operating systems, passwords, computer security
CR Categories: 2.41, 4.35

Introduction

Password security on the UNIX (a trademark of Bell Laboratories) time-sharing system [3] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we

tion of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e., prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so-called "super-user" password on the UNIX system, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual authorized access and accidental disclosure.

UNIX Password File

- Traditionally, Unix stored 'encrypted' password in the file */etc/passwd*, which is world-readable
- Example:

```
root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
uucp:OORoMN9FyZfNE:4:4::/var/spool/uu:/usr/lib/uucp/uucico
rachel:eH5/.mj7NB3dx:181:100:Rachel Cohen:/u/rachel:/bin/ksh
arlin:f8fk3j1OIf34.:182:100:Arlin Steinberg:/u/arlin:/bin/csh
```

- Is the problem of password storage security solved?
- Not quite. An attacker can launch an ***off-line*** password cracking attack
  - Try different passwords, hash them and compare result with any of the entries in the password file. Continue until there is a match.
- → 'Encrypted' or hashed passwords need to be hidden
  - "Shadow passwords"
  - Store password hashes in a separate file with restricted access, e.g. */etc/shadow*
  - *For practical reasons, access to /etc/passwd file cannot be restricted. A lot of utilities rely on other information stored there, e.g. user IDs, Groups IDs, …*

SALT

- What if two users happen to choose the same password?
  - Let’s assume shadow passwords are not used
- They will detect this by looking at /etc/passwd and will be able to access each other’s account.
- Unix originally used 12 random bits in a modified DES algorithm to prevent this.
  - This is called ‘SALT’

Modern UNIX Password Files

- /etc/passwd:

```
root:x:0:0:root:/root:/bin/ksh
uucp:x:66:1:uucp:/var/spool/uucp:/sbin/nologin
rachel:x:1181:100:Rachel Smith:/home/rachel:/bin/ksh
arlin:x:1652:100:Arlin Federics:/home/arlin:/bin/csh
```

- /etc/shadow:

```
root::$6$Vs/5TFoOyPArcE.U$PNN9jt56NSwCLuTTOviSiWXwoHkw1X4CK1zF
   Dqci8d3NN/CV6Rt7xYooY0ecXcA6LIFTDfp6CKhZklmxKZgzg.:14542:0:
   99999:7:::
uucp:*:14344:0:99999:7:::
rachel:!!:14544:::::
arlin:$1$/80DehIe$.11g0DCK3CmY/UtX.mr6c/:14687:::::
```

Symmetric Cryptography

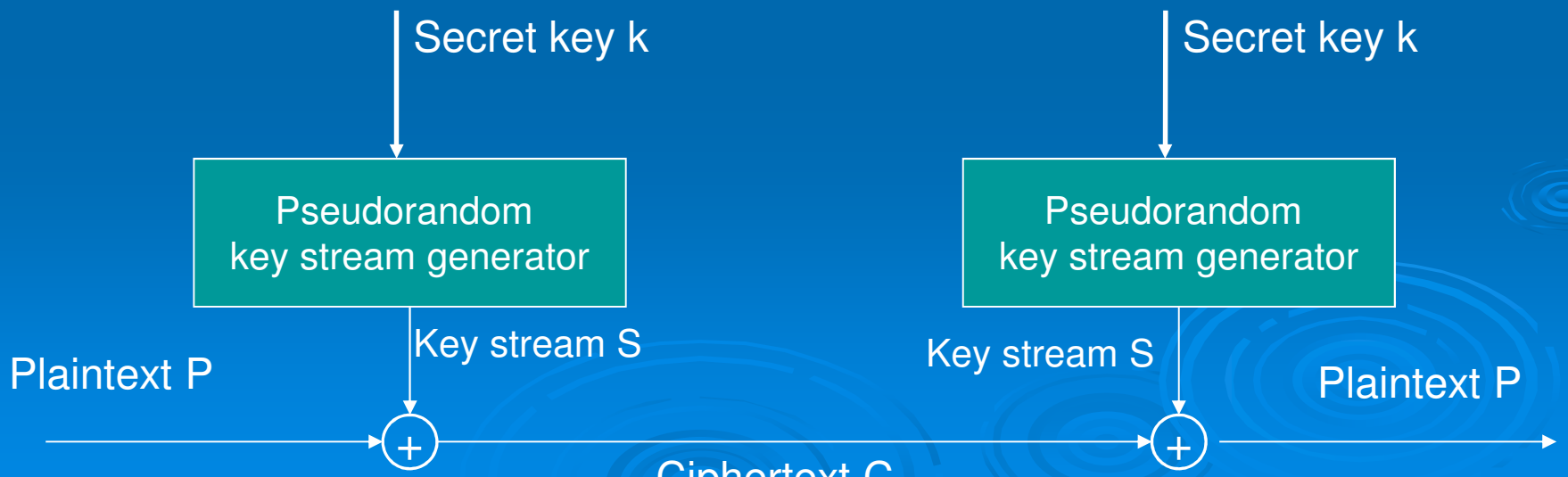
Modern Symmetric Ciphers (Secret Key Ciphers)

Block Ciphers vs. Stream Ciphers

- So far all, the ciphers we considered were so-called Block Ciphers (except for the one-time pad)
- Block Cipher
  - Works on fixed-size plaintext blocks (64bits, 128 bits, etc.) and produces blocks of ciphertext of the same size
  - Most common ciphers today are block ciphers
- Stream Cipher
  - Works on smaller units of plaintext (bits or bytes)
  - Generates *pseudorandom* key stream
    - Difference to OTP, which uses *random* key stream
  - Encryption is typically done via XOR-ing key stream with plaintext
    - Similar to one-time pad, only that key stream is not truly random here
  - Key stream is typically generated via feed-back mechanism using shift-registers

Stream Ciphers

- Work like one-time pad
- Generate pseudorandom key stream S
  - S is a function of a secret key k
  - Expand k into a arbitrarily long key stream
- Encryption: C = P XOR S (bit-wise XOR)
- Decryption: P = C XOR S
- No perfect security since key stream S is not truly random and will eventually be repeated



Stream Cipher - RC4

- “Ron’s Code”, Ron Rivest, The ‘R’ in RSA
- Most widely used stream cipher
  - MS Word, Excel, Wireless LANs (WEP)…
- Very simple and elegant
  - Can be implemented in few dozen lines of code
- Secure?
  - Some vulnerabilities have been discovered in the past few years. AES is the better choice.

RC4-based cryptosystems

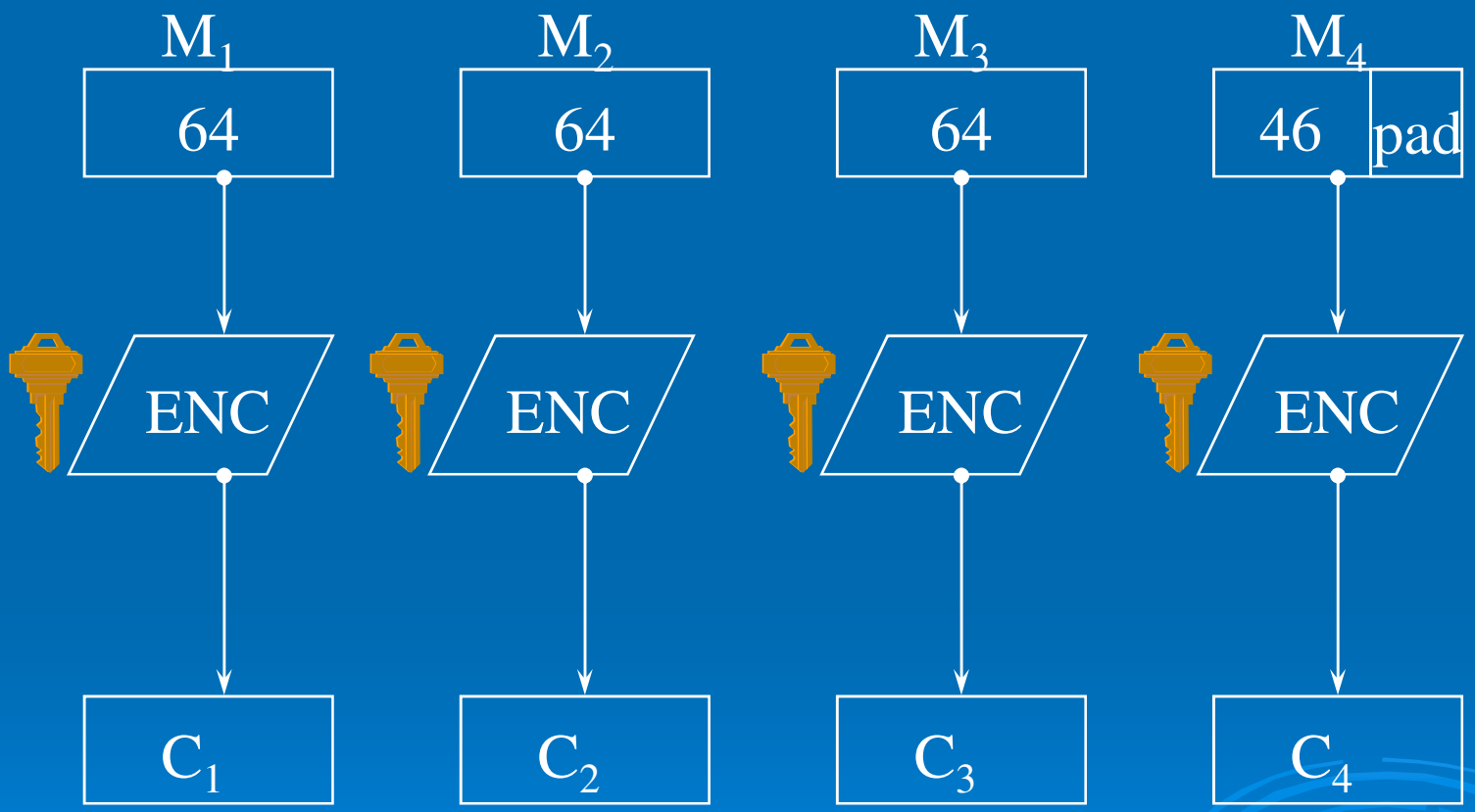
- WEP
- TKIP (WPA/WPA2 option)
- BitTorrent protocol encryption
- Microsoft Point-to-Point Encryption
- Secure Sockets Layer (option)
- Secure shell (option)
- Remote Desktop Protocol
- PDF

Any questions so far?

Encryption Modes of Block Ciphers

- Assume you have a piece of hardware/software that implements AES
  - Input 128-bit plaintext
  - Output128-bit ciphertext
- How do you use this to encrypt a stream of data, e.g. a large file?
  - → Cipher Modes

Electronic Code Book Mode (ECB)



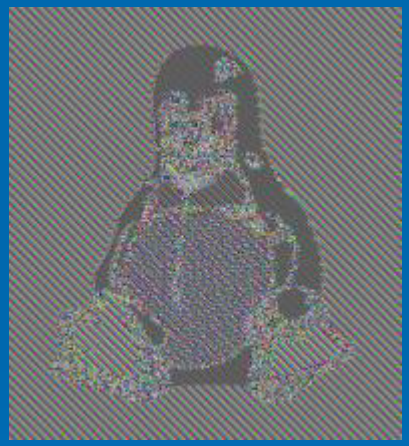
Problems with ECB

- Weaknesses?
- What are possible attacks against this mode?

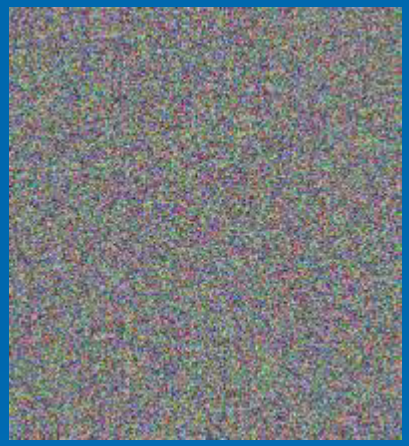
- The same plaintext block always results in the same ciphertext block (no dependencies between blocks!)
- → Replay attack
  - E.g. resend a block that says “Transfer $1000 from Bob’s to Trudy’s account”.
- → Ciphertext can be reordered → reordered plaintext
- → ‘Dictionary attack’
  - Build a dictionary of all plaintext-ciphertext block pairs
  - Not feasible for large block size

Problems of ECB Encryption

Original



Encrypted in ECB mode



Encrypted in secure Mode
(CBC, CFB, OFB,…)

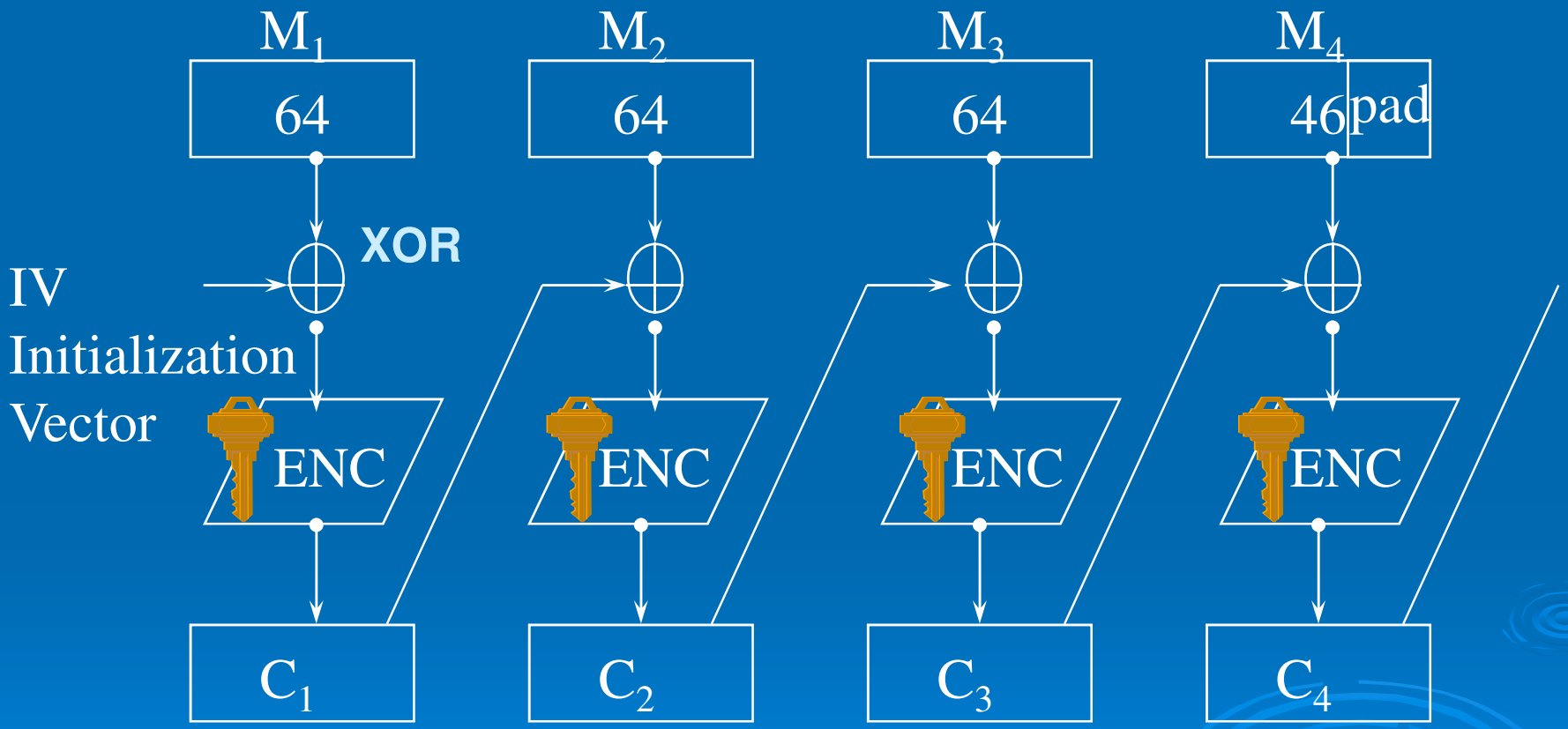
ECB Summary

- Advantages
  - Simple
  - Fast implementation
    - Can use pipelining/parallelism
  - Error propagation
    - A bit error in a block only affects the decryption of this particular block
- Problems
  - Vulnerable to insertion, replay, reordering, dictionary attacks
  - → ECB is not considered secure and should be avoided

Solution

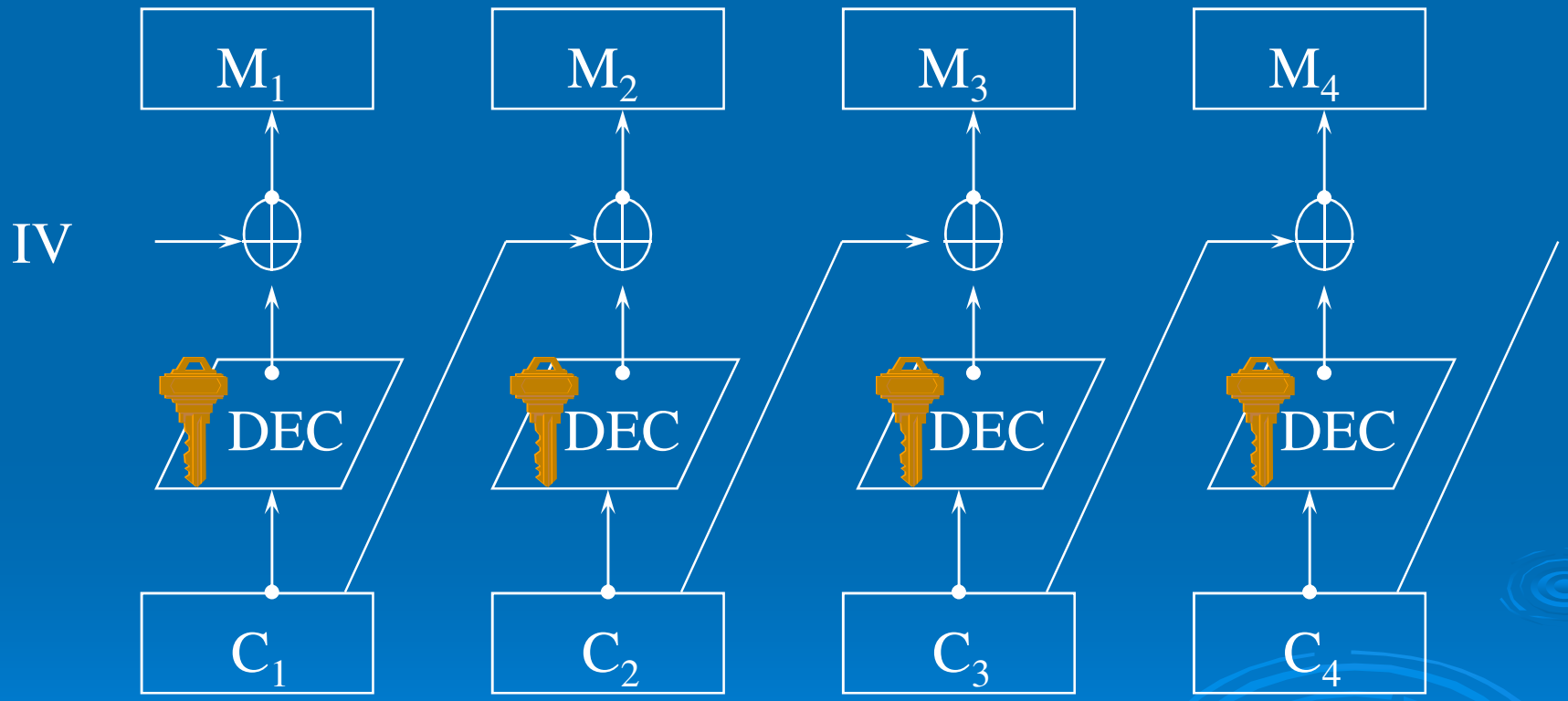
- Feedback mechanism
  - The result of the encryption of a previous block is fed back into the encryption of the current block.
  - Ciphertext blocks depend on plaintext block but also on previous ciphertext blocks
  - Different ways to use feedback
    - CBC
    - CFB
    - OFB

Cipher Block Chaining Mode (CBC) Encryption



Same input results in different output with high probability.
How does Decryption work?

CBC Decryption



Initialisation Vector

- IV does not need to be secret
  - Just a dummy ciphertext block to start with
- IV should be changed frequently
  - Encrypting the same plaintext with the same key and same IV still results in same ciphertext

CBC

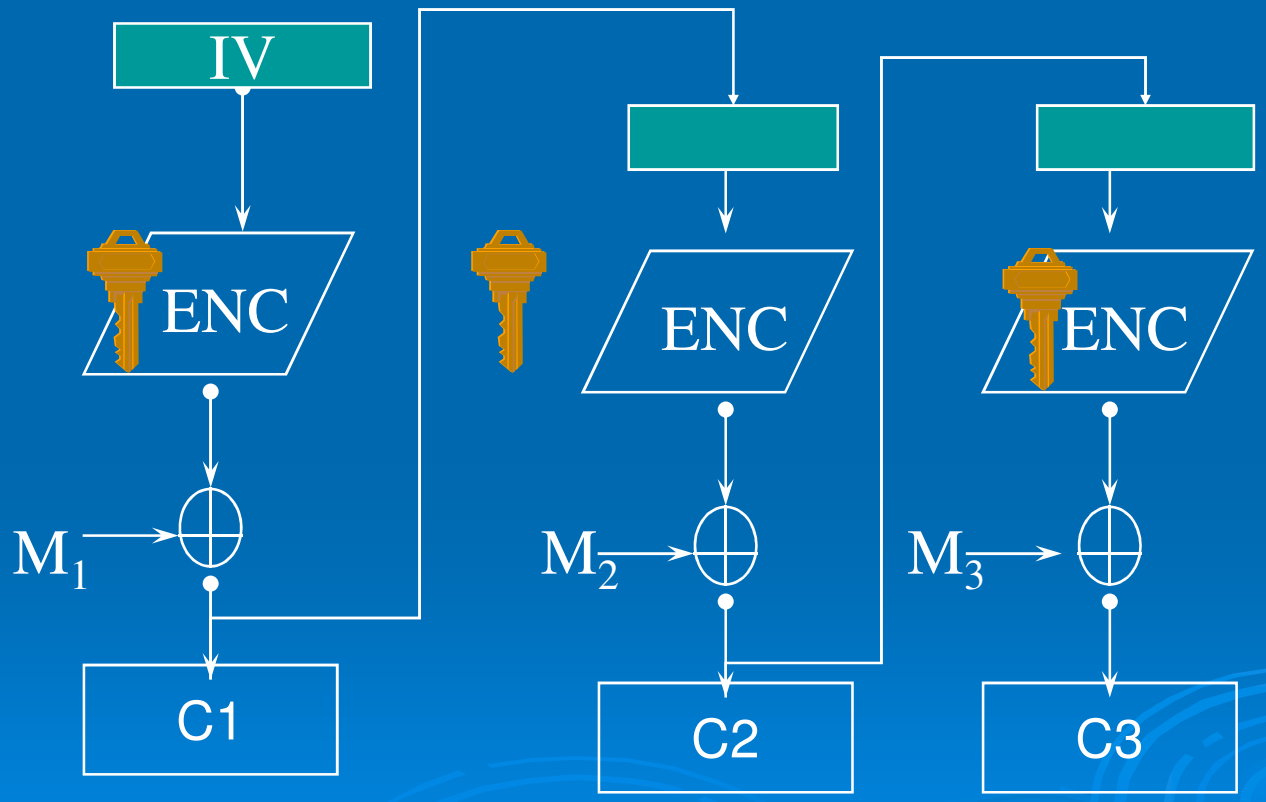
- Each block of ciphertext depends on current plaintext block, previous ciphertext block and IV
  - Prevents:
    - Reordering Attack
    - Insertion Attack
    - Replay Attack
    - Dictionary attacks

CBC - Handling of Transmission Errors

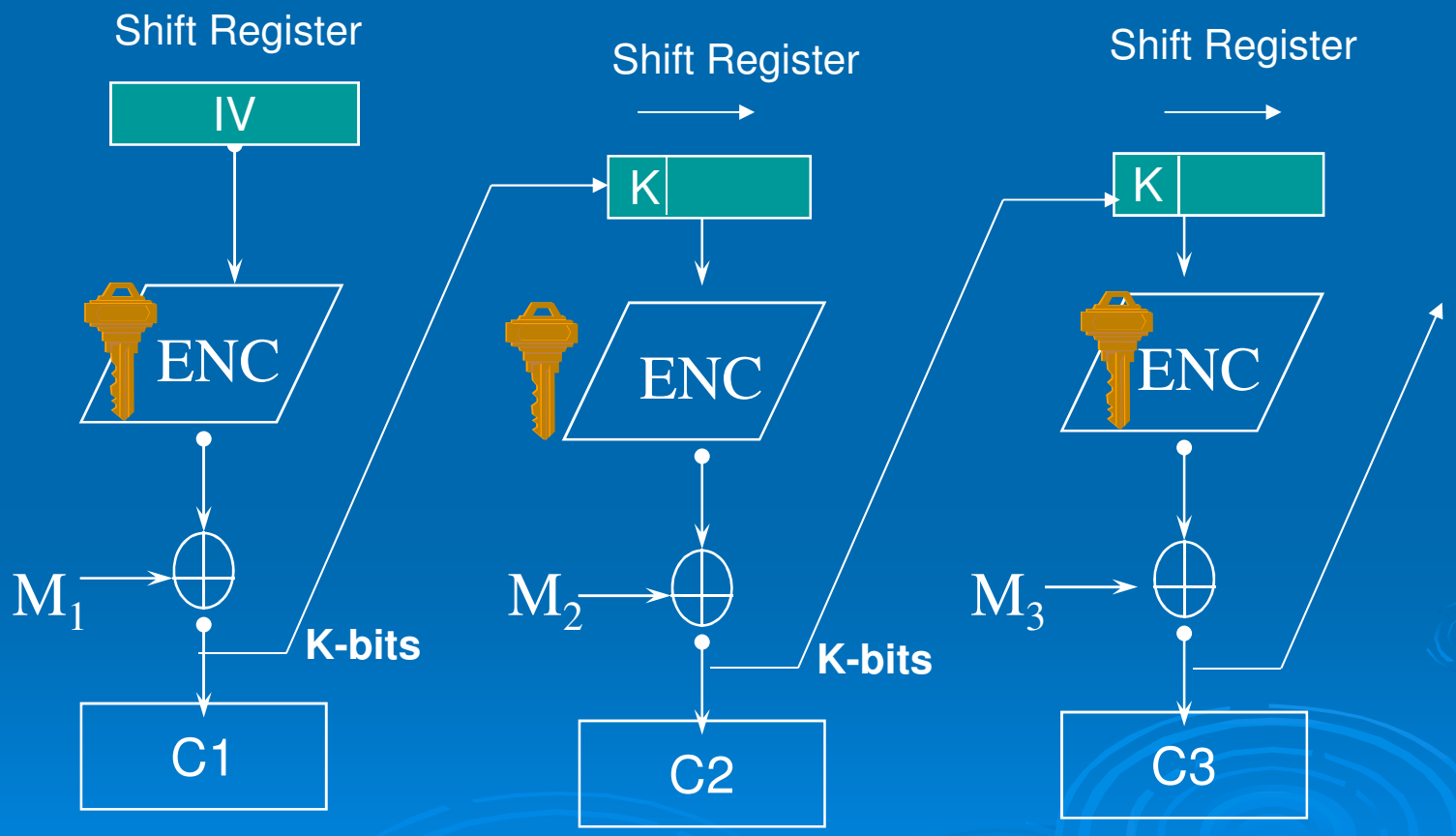
- Error Propagation:
  - Loss of a complete ciphertext block
    - Results in only one plaintext block being garbled. CBC re-synchronises after that!
  - Bit error in a ciphertext block
    - Results in the corresponding plaintext block being garbled
    - The next plaintext block has a single bit error at the same position as the bit error in the corresponding ciphertext block
    - Attacker could use this to achieve predictable bit changes in plaintext blocks
    - Further plaintext blocks are not affected
  - → CBC is self-recovering from bit errors
- Implementation: Can encryption of multiple blocks be done in parallel ?
  - No, en/decryption of block i depends on block en/cryption of block i -1
- CBC is the most commonly used cipher mode
- Is supported by all IPsec and TLS/SSL implementations

Cipher Feedback Mode (CFB)

Encryption

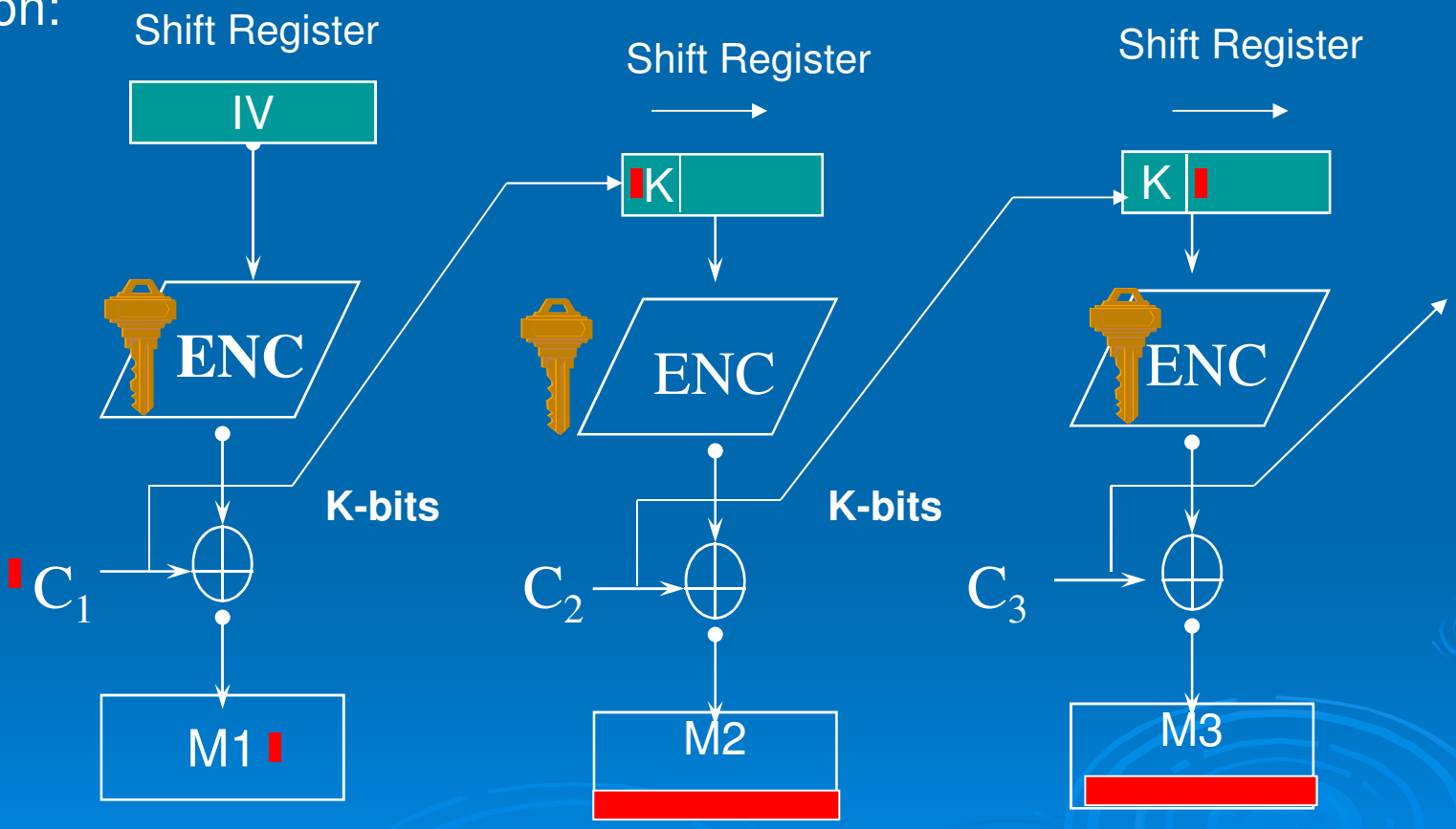


General k-bit Cipher Feedback Mode (CFB)



CFB – Error Propagation

Decryption:



Note: Encryption function of Cipher is used here as well

CFB Error Propagation

- Q: For a block size of 64 bits and k=8, how many blocks of plaintext are affected by a single bit error in the ciphertext
- A: 9 (the current block plus the 8 following blocks)
  - It takes 64/8=8 times for the erroneous bit to fall off the other end of the shift register

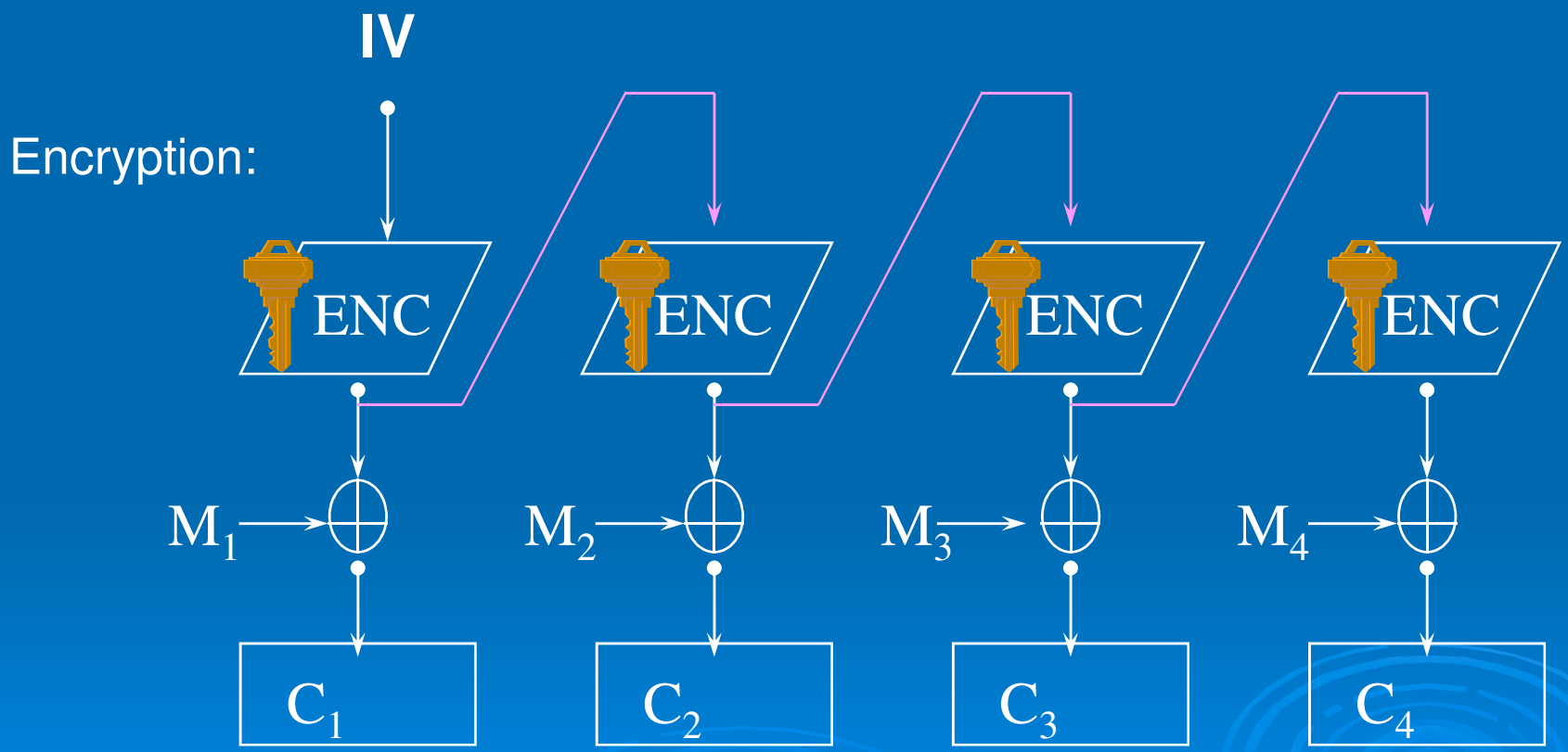
CFB Synchronisation

- CFB can recover from whole blocks being deleted from ciphertext stream
  - In the same way as it handles bit errors
  - Self-synchronising at block level, similar to CBC

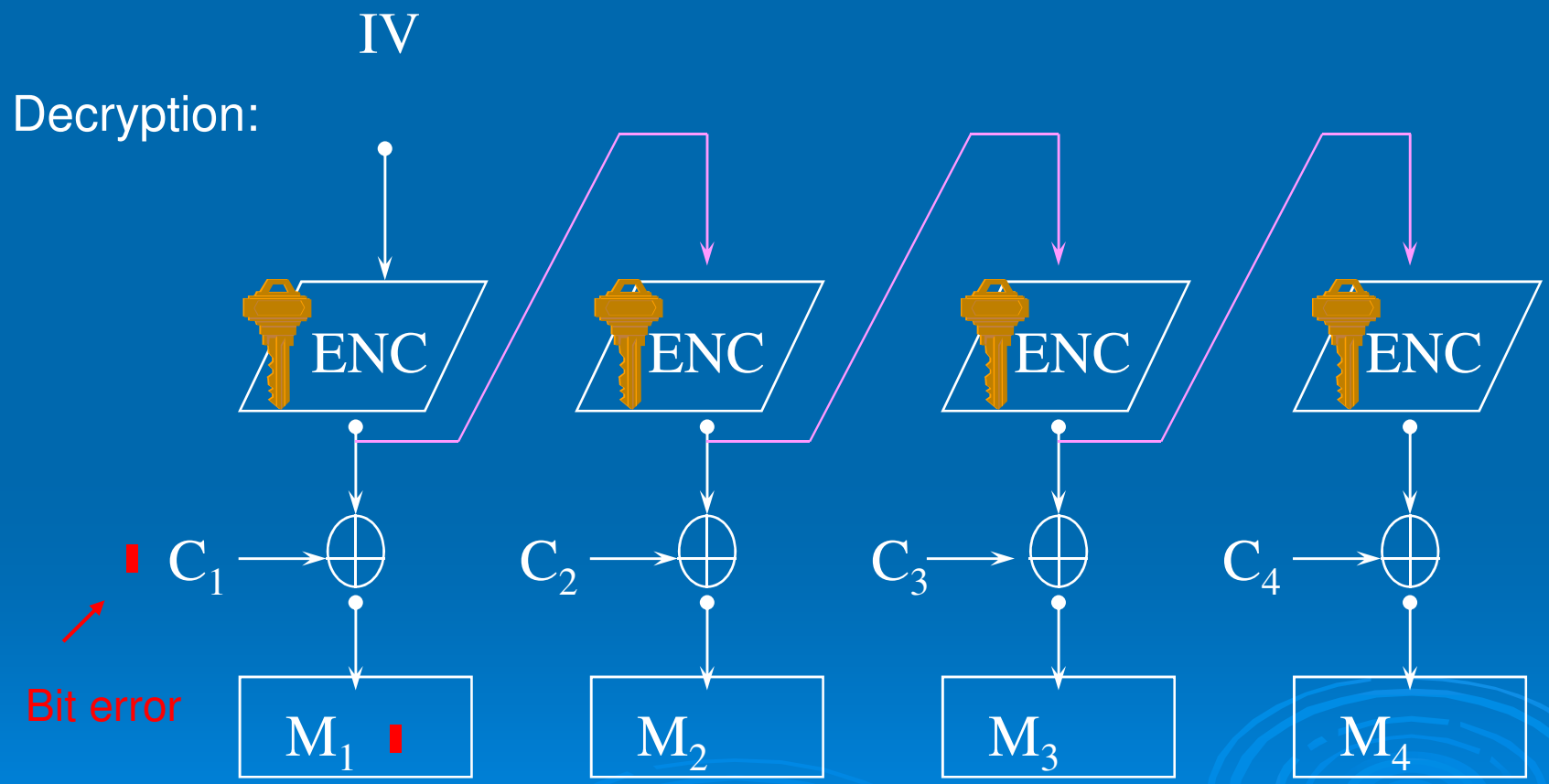
Output Feedback Mode (OFB)

IV should be unique but does not need to be secret

Uses a block cipher to in “stream cipher” mode
Similar to one-time pad



Output Feedback Mode (OFB) Error Propagation



Output Feedback Mode (OFB)

- Advantages
  - No error propagation (or error extension) as in CBC
    - 1 bit error in ciphertext → 1 bit error in plaintext
    - Useful for voice or video for example
  - Allows pre-computing of pseudo-random stream
  - XOR can be implemented very efficiently
    - Works like a Stream Cipher
  - Does not need a full block to start encryption.
    - Useful for terminal applications
- Problems
  - Not self-synchronising
    - Cannot recover from loss of entire ciphertext blocks

Any questions?

Asymmetric Cryptography

Encryption – A Mechanical Analog

Mechanical analogue of public key cipher

Mechanical analogue of symmetric cipher

Public Key Cryptography

- Invented in the 1970s
- The major breakthrough in cryptography in the last 2,000 years or so.

- Main idea
  - Encryption key ≠ Decryption key

- Benefit
  - Can make encryption key public
  - Solves key distribution (management problem)
    - At least partially, as we will see later

Public Key Cryptography

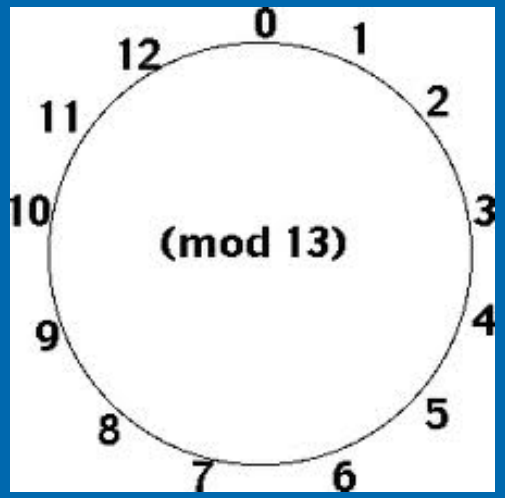
- Radically new approach
  - Famous Paper: Whitfield Diffie, Martin E. Hellman, “New Directions in Cryptography” (1976)
    - IEEE Transactions on Information Theory
    - (idea independently developed by Martin Merkle)
- Encryption Key ≠ Decryption Key
  - “asymmetric cryptography”
- Makes key management a lot easier
  - Encryption key can be made public (“public key”)
    - Allows encryption but not decryption (“one-way operation”)
    - No secure channel is needed (confidentiality)
      - (We will see later that we still need an “authenticated channel”)
    - Only the owner of the corresponding “private-key” can decrypt

Public Key Cryptography

- The basic idea is simple, but..
  - How do we find such an asymmetric cipher?
- Diffie and Hellman outlined the basic idea of Public Key crypto
- Definitions used by Diffie and Hellman
  - **One-way function**: 'easy' to compute, 'hard' to compute the inverse
    - Not new
  - **New Concept**: **Trapdoor one-way function** ("mathematical padlock")
    - Function is one-way, except if secret 'trapdoor' is known
- Diffie and Hellman did not find a trapdoor one-way function
  - But they found a one-way function, and showed how it can be used to establish a secret key between two parties
- First (publicly known) practical Public Key Cryptosystem by Ron Rivest, Adi Shamir, and Leonard Adleman.
  - → **RSA**

Modular Arithmetic

- Most Public key algorithms use modular integer arithmetic. Numbers 'wrap around'.
- Examples
  - (9+8) mod 13 =?
    - = 4
  - (3*6) mod 13 = ?
    - = 5



- *Useful properties*
  - *(a + b) mod n = (a mod n + b mod n) mod n*
  - *(a * b) mod n = (a mod n * b mod n) mod n*
  - *(a + c*n) mod n = a mod n (for all c)*
    - *These numbers are 'congruent'*
- *Another Example:*
  - *(9 * 17) mod 7 = ?*
  - *9 mod 7 * 17 mod 7 = 2 * 3 mod 7 = 6*

By the way, XOR is addition (and subtraction) modulo 2

Modular Exponentiation

- 'Normal' Exponentiation Rules apply:
  - $(g^x)^y = (g^y)^x = g^{xy}$
  - $g^x * g^y = g^{x+y}$

- $7^4 \bmod 9 = (7*7*7*7) \bmod 9 = ?$
- $[\, 7^2 \bmod 9 * 7^2 \bmod 9] \bmod 9 = (4 * 4) \bmod 9 = 7$
- $7^8 \bmod 9 = \ ?$
- $7^8 \bmod 9 = 7^{(4+4)} \bmod 9 = 7^4 \bmod 9 \ * \ 7^4 \bmod 9 =$
- $7*7 \bmod 9 = \ 49 \bmod 9 \ = 4$
- $7^{12} \bmod 9 = \ ?$
- $7^{12} \bmod 9 = 7^{(4+8)} \bmod 9 = 7^4 \bmod 9 \ * \ 7^8 \bmod 9 = \ (7 * 4) \bmod 9 = 1$
- ...
- Modular Exponentiation can be implemented very efficiently via the **"square-and-multiply"** algorithm with running time $O(\log_2 n)$, with n being the size of the exponent.
  - See tutorial for details

Discrete Logarithm

- The inverse of modular exponentiation is the **"Discrete Logarithm"**
  - $7^{4} \pmod 9 = 7$
  - $7^{?} \pmod 9 = 7$
  - $\log_{7} 7 \pmod 9 = 4$
- What about the following?
  - $\log_{3} 2 \pmod 4 = x = ?$
  - $3^{x} \pmod 4 = 2$
- Let's try to find the answer by trial and error:
  - $3^{0} \pmod 4 = 1$
  - $3^{1} \pmod 4 = 3$
  - $3^{2} \pmod 4 = 1$ (Problem: $\log_{3} 1 \pmod 4$ has two solutions !!!)
  - $3^{3} \pmod 4 = 3$ (Problem: $\log_{3} 3 \pmod 4$ has two solutions !!!)
  - ➔ $\log_{3} 2 \pmod 4 = ?$ Has no solution!

Discrete Logarithms

- Under certain conditions, it is guaranteed that the discrete logarithm $log_a\ c\ mod\ n$ has a unique solution for all c
  - *n* needs to be a prime number
  - *a* needs to be a "*generator" (of the 'Cyclic Group of integers modulo n')*
    - *a* is a generator if by computing a^b for b = {1,2…n-1}, all elements 1,2 …n-1 are "*generated" .(This means that for all c the log exists.)*
- *Example: n = 5, a = 3*
  - *Is a=3 a generator?*
    - $3^1\ mod\ 5 = 3$ → $log_3\ 3\ mod\ 5 = 1$
    - $3^2\ mod\ 5 = 4$ → $log_3\ 4\ mod\ 5 = 2$
    - $3^3\ mod\ 5 = 2$ → $log_3\ 2\ mod\ 5 = 3$
    - $3^4\ mod\ 5 = 1$ → $log_3\ 1\ mod\ 5 = 4$
    - *a is a 'generator' (of the multiplicative group of integers modulo 5)*
  - Number Theory tells us that *a* is a generator if gcd(a, n-1) = 1
  - *So we can directly check this, without trial and error.*
    gcd(a, n-1) = gcd(3, 4) = 1 → a is a 'generator'

Example

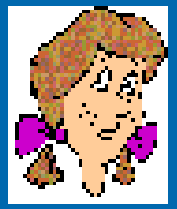
- $\log_5 6 \bmod 7 = ?$
- $5^x \bmod 7 = 6 \rightarrow x = ?$
- *Trial and error:*
  - $5^1 \bmod 7 = 5$
  - $5^2 \bmod 7 = 4$
  - $5^3 \bmod 7 = 6$
- $\log_5 6 \bmod 7 = 3$
- Is this method of computing discrete logarithms efficient?
- How about this:
  - $\log_{324554332333} 7876665 \bmod (357425491988726172913535086566266425 67) = ?$

Discrete Logarithm Computation

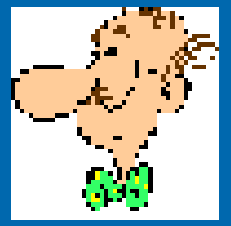
- No efficient algorithm is known to compute discrete logarithms for large *n*.
  - Since no one has found such an efficient algorithm, it is assumed that none exists, but no one knows for sure
    - 'Diffie-Hellman-Pohlig Conjecture' (not a Theorem, no proof)

- *→ Modular exponentiation is a one-way function*
  - *(not a trapdoor one-way function!)*

Diffie-Hellman Key Agreement Protocol



- Alice and Bob initially agree on a large prime number *p* and a generator *g*
  - → *discrete logarithms exist*
- *All computations are done modulo p*



- Alice chooses random number x
- Alice computes $X=g^x$ and sends it to Bob

g^x →

← g^y

- Bob chooses random number y
- Bob computes $Y=g^y$ and sends it to Alice

- Alice computes $K_{AB} = Y^x = (g^y)^x \bmod p = g^{xy} \bmod p$

- Bob computes $K_{AB} = X^y = (g^x)^y \bmod p = g^{xy} \bmod p$

- Eve is listening on the insecure channel and sees g^x and g^y, but since she cannot calculate discrete logarithms, she does not know x or y
- → She cannot compute the secret key K_{AB}

- Is there a possible attack against this protocol?

Man-in-the-Middle Attack

- Messages between Alice and Bob are not authenticated
- An active Attacker can pretend to be Alice to Bob, and can pretend to be Bob to Alice.
- Problem
  - Lack of authentication
- Illustration:
- http://dropbox.eait.uq.edu.au/uqmportm/coms3000/ManInMiddle-DH.html

Any questions so far?

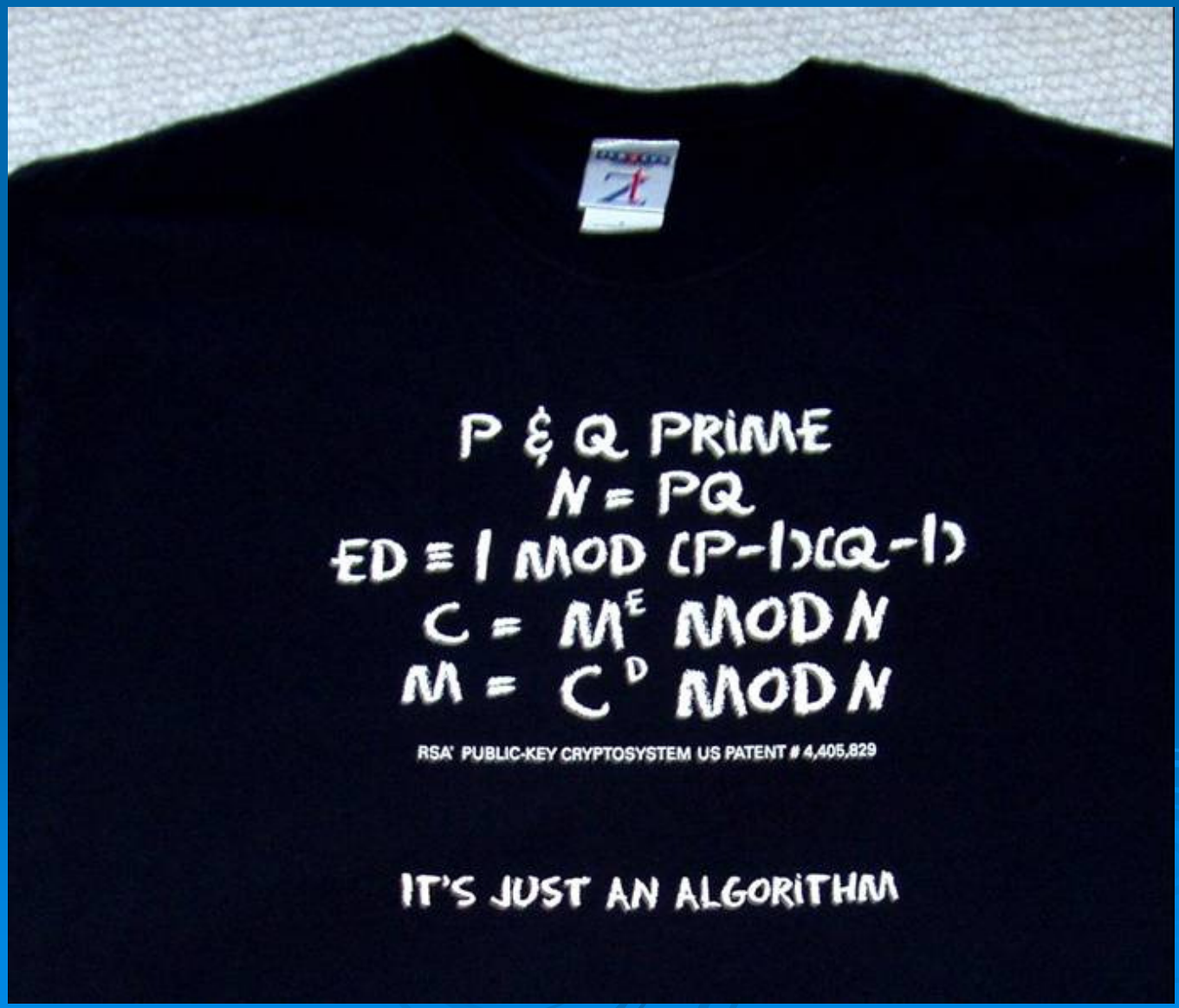
RSA

- The Diffie-Hellman protocol allows the establishment of shared secret keys, but it is not a public-key encryption algorithm
  - The “Discrete logarithm” problem provides us with a one-way function, but not a **trapdoor one-way function** (“mathematical padlock”)
- First (publicly known) Public-Key Cipher
  - RSA
  - Inventors:
    - Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman

‘Going Public’

- Simon Singh’s videos on the invention of Public Key Cryptography
  - *https://www.youtube.com/watch?v=_ZTWLAqYf9c*
  - *https://www.youtube.com/watch?v=oR0_LPbWxe4*

RSA - How it works



RSA – How it works

- We use modular arithmetic, *mod n*
  - *n* is called the **modulus**
- Bob chooses two large (> 100 digits) primes p and q and computes
  - n = p * q
- Bob computes z = (p-1) * (q-1) ("Euler's Totient function")
- Bob chooses exponent e that has no common factor with z
  - ("z and e are relatively prime"), or gcd(e, z) = 1
- Bob keeps p and q secret, but sends Alice (n, e)
  - (n, e) is Bob's public key
- Alice can encrypt a message m as follows:
  - $c = m^e \bmod n$ (Modular exponentiation is easy!)
- Bob computes d such that $c^d = m \bmod n$
- $c^d = (m^e)^d \bmod n = m^{ed} \bmod n = m$
- d is the secret key that allows decryption (trapdoor)

RSA Decryption - Finding ***d***

- Problem: How does Bob find the secret key *d* that allows decryption?
  - $(m^e)^d \bmod n = m$
- We can find d if the prime factors p and q of n are known
  - Find d such that $e*d \bmod z = 1$, $z=(p-1)(q-1)$
    - There exists an efficient algorithm for this: "**Extended Euclid's Algorithm**"
- There is no known efficient way to find d for large n without knowing p and q, the prime factors of n
  - "Factoring" is considered a "hard problem", in the same way as computing Discrete Logarithms is considered a hard problem
- There is no known efficient way to invert the encryption $m^e \bmod n$, without the trapdoor d.
- Bob can now discard p and q, but must not reveal them
- Exponentiation $m^e \bmod n$ is a trapdoor one way function with *d* being the trapdoor

RSA Security

- It has been proven that breaking RSA is equivalent of solving the age-old factoring problem.
- This is not impossible, but unlikely, given not just cryptographers, but also mathematicians have tried unsuccessfully for centuries to solve this problem.

RSA Example (very small modulus)

- Bob chooses p=3 and q=11
  - (far too small to be secure)
- n = p * q = 11 * 3 = 33
- z = (p-1) * (q-1) = (3-1)*(11-1) = 20
- e=?
  - (e cannot have a common factor with z =20)
  - e = 3, or 7, 9, 11, …
- d=? (e*d mod z = 1)
  - For large numbers → We can use the Extended Euclid's Algorithm, if we know p and q
  - For small numbers, we can do this via trial and error.
    - d=1: 3*1 mod 20 = 3
    - d=2: 3*2 mod 20 =2
    - .
    - .
    - **d=7: 3*7 mod 20 = 1**
- Bob's Public Key: (n, e)= (33,3)
- Bob's Private Key: (n, d)= (33,7)

RSA

1) choose 2 primes p and q, n=p*q
2) z = (p-1)(q-1)
3) Choose e so that e and z are relatively prime
4) Compute d so that e*d mod z =1

Public Key: (n, e)
Private Key: (n, d)

Encryption: $c=m^e \bmod n$
Decryption: $m=c^d \bmod n$

RSA Key Length

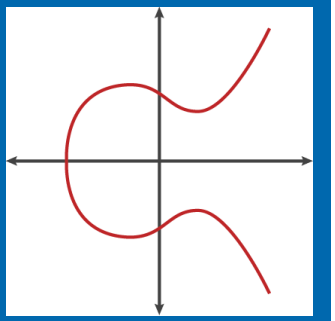
- Which one is more secure?
  - 128 bit AES key
  - 1024 RSA key
- According to RSA Security (2003)
  - 1024 bit RSA key is equivalent in strength to 80 bit key of a symmetric cipher
    - Don't need to brute-force 1,024 key space as in AES, need to factor 1,024 bit integer, which is still hard, but not as hard as 'brute forcing' a 128-bit key.
    - 768-bit RSA key has been broken
    - There is a $100,000 reward for breaking a 1,024 bit RSA key
      http://en.wikipedia.org/wiki/RSA_Factoring_Challenge
  - 2048 bit RSA ~ 112 bit symmetric key
  - 3072 bit RSA ~ 128 bit symmetric key
    - http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf
    - Section 5.6.1
- Security of RSA relies on the fact that factoring is hard
- Integers can be efficiently factored, i.e. RSA can be broken, with a quantum computer, using Shor's Algorithm. However, we probably won't have to worry about the practicality of **quantum computers** for some time (?).

Public Key Crypto Systems

- Most Public Key systems rely on either the factoring or discrete log problem being hard, i.e. that there is no efficient algorithm for this.

- Example:
  - *El Gamal* public key encryption system is based on the discrete logarithm problem
  - http://en.wikipedia.org/wiki/ElGamal_encryption

Elliptic Curve Cryptography (ECC)



- Public key cryptography on special mathematical structure called *Elliptic Curves*
- Benefit
  - Best known algorithms for computing discreet logarithms or factoring in ECC are much slower.
  - → Can achieve same level of security with shorter key length, compared to standard RSA, Diffie Hellmann, etc.
  - http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

Public Key Cryptography

- Public Key Cryptography solves the key distribution problem, but …
- Public Key encryption/decryption is orders of magnitudes slower than symmetric key cryptography
  - Modular exponentiation with very large numbers is significantly more expensive than the type of operations used in symmetric ciphers
- Solution?
- → Hybrid Systems
  - Public Key Cryptography is only used to establish symmetric secret 'session keys'
  - For example, Alice could encrypt a 128-bit AES key with Bob's public key (signed with her private key)
  - Used in protocols such as TLS (and previously SSL)

RSA – Interesting Property

- Encryption and decryption are essentially the same operation and can be applied in any order ('commutativity')
  - $(m^e)^d \bmod n = (m^d)^e \bmod n = x^{ed} \bmod n$
- For example, Bob could encrypt a message with his private key *d,* and Alice (and everybody else) could decrypt it with Bob's public key *e*
  - Bob computes $c = m^d \bmod n$
  - Alice computes $m = c^e \bmod n$
- How could this be useful?
  - It's a **Digital Signature**
  - Everybody can decrypt the message using the public key and verify that it is a valid message (need some redundancy)
  - Only someone knowing the corresponding private key could have encrypted message so that it decrypts into a valid message
  - → Provides Authentication (and also Integrity)

RSA Signatures

- If Alice encrypts a message *m* with her own private key, who will be able to decrypt it?
  - Everybody, her 'public key' is public
  - → **public key signature**
  - Anyone decrypting it with Alice's public key will know that *m* must have been encrypted with the corresponding private key.
  - Presumably Alice is the only person who knows her private key, therefore, Alice must have been the person who encrypted *m*.

RSA Signatures

- How can we digitally sign a large document *f*, say a 600MB file?
- Remember, modular exponentiation with large numbers is slow, about 1,000 times slower than secret-key algorithms
- Solution?
  - Use a cryptographic hash function h()
  - Compute h(f) → 128 bit value (hashing is cheap)
  - Sign (encrypt with private key) h(f) instead of the whole file *f*
    - $s = (h(f))^d \bmod n$
  - Send file *f* plus signature *s*
- How can receiver verify signature?
  - Decrypt the received signature s_R with the public key → $s_R^e \bmod n = h_R(f)$
  - Compute the hash of the file h(f)
  - If the two values are the same, i.e. if $h(f) = h_R(f)$, the signature is valid

- → It is guaranteed that the file
  - was sent by the owner of the corresponding private key
    - Provides Authentication and Non-repudiation
  - has not been tampered with (Integrity)

- This obviously relies on the fact that an attacker cannot find a hash collision, i.e. another file f2, so that h(f) = h(f2).

Any questions so far?

Public Key Cryptography

- Can be used for Confidentiality
  - Encryption
- Can be used for Authenticity/Integrity
  - Digital Signatures
- Makes key distribution problem a lot easier
- … so far so good.

Consider the following …

- Trudy sends the following email to all of Alice's friends, including Bob:

Dear Friend,
This is my new public key: 23245675444…
Cheers
Alice

- What is the problem here? (Two things)
  - Bob might use the key to send a secret message to Alice. He encrypts his message with this key and sends it to Alice. Trudy eavesdrops and only she can now decrypt the message because only she has the corresponding secret key.
    - Attack on Privacy
  - Trudy can send a digitally signed message to Bob with Alice's signature. If Bob believes that the key received from Trudy is Alice's, he will be convinced that digital signature is correct and the message is from Alice.
    Attack on Authenticity

Authenticity of Public Keys

- For Public Key Cryptography to be secure we need to make sure public keys are authentic.
- We need trust in the '**identity – key**' binding

- How can we do that?
  - A trusted third party vouches for the identity-key binding
  - **Trent** is trusted by Bob and Alice. He writes the following document and digitally signs it with his private key:
    - "I, Trent, guarantee that Alice's public key is 23245675444…" Trent's Signature.
  - This document is called a **digital certificate**.

PUBLIC KEY CERTIFICATES

PUBLIC KEY CERTIFICATE

```
RSA Public Key: (1024 bit)
            Modulus (1024 bit):
                00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                e8:35:1c:9e:27:52:7e:41:8f
Exponent: 65537 (0x10001)
```

I, Trent, hereby certify that the above public key belongs to

Alice

27/10/2014

Signature

Date

Public Key Certificates

- While the certificate on the previous slide is certainly beautiful, is it practical?
  - No, we need an electronic, digital version.
  - So, we need a digital version of Trent's signature → Digital Signature
    - e.g. using RSA

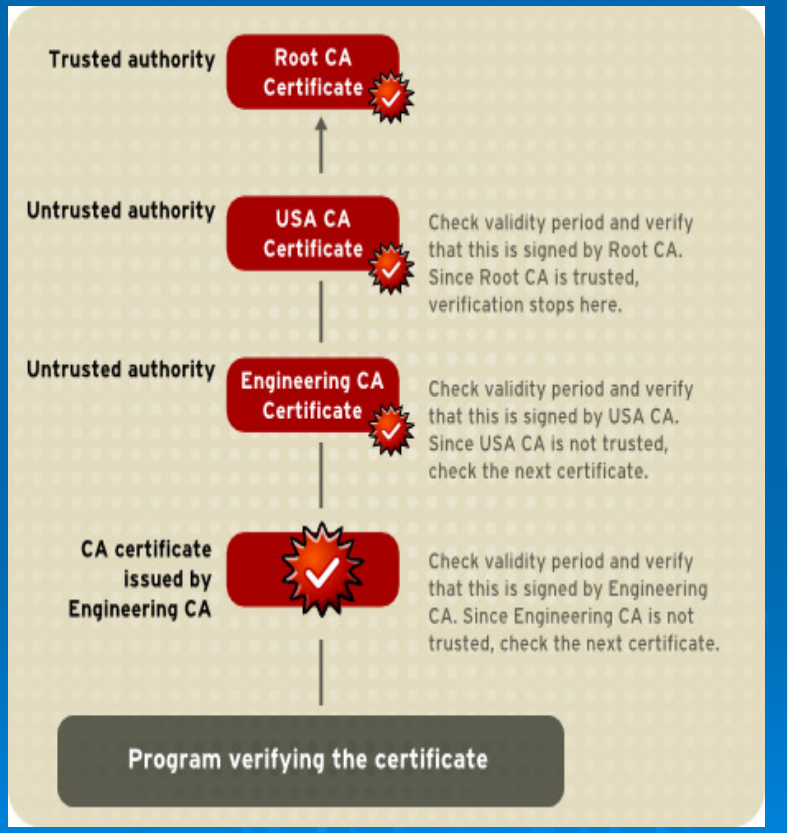
- If we assume Trent is trustworthy, is the problem solved?
- We also need to verify Trent's signature on the certificate.
- How do we do this?
  - We need Trent's public key for that.
  - How can we trust the key we have is authentic, i.e. really Trent's and not a 'fake' sent by Trudy?
  - We need a public key certificate for Trent's public key as well
  - Maybe another third party can vouch for it…
  - (→ certificate chain)

  - We have a "chicken and egg" situation here.
  - This 'bootstrapping' of trust cannot be solved with cryptography

Public Key Certificate Chains

- Public Key Certificate
  - **Links together Identity and Public Key**
- Signed by a trusted third party, called a ***Certification Authority*** or ***CA***
- *CAs can have hierarchical structure*
- Certificate Chains →
- At the top is the *Root Certificate*
  - self-signed
  - Serves as a 'trust anchor'
  - Trust is provided (bootstrapped) via other means, e.g. via pre-installation in web browser or certificate store

https://access.redhat.com/documentation/en-US/Red_Hat_Certificate_System/8.1/html/Deploy_and_Install_Guide/How_CA_Certificates_Establish_Trust-Verifying_a_Certificate_Chain.html

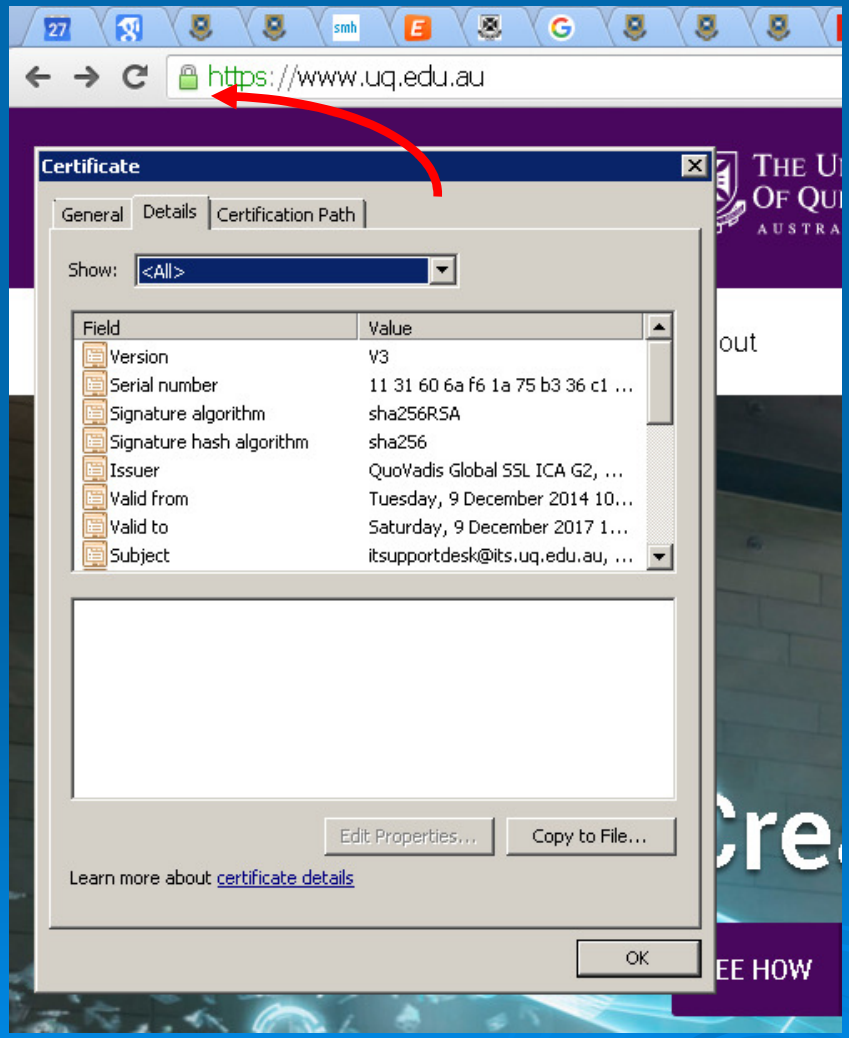


X.509 Certificates

- X.509: Public Key Certificate Standard issued by ITU-T
  - Defines format of certificates
  - Most commonly used format of certificates
- More details:
  - http://en.wikipedia.org/wiki/X.509
  - http://tools.ietf.org/html/rfc5280

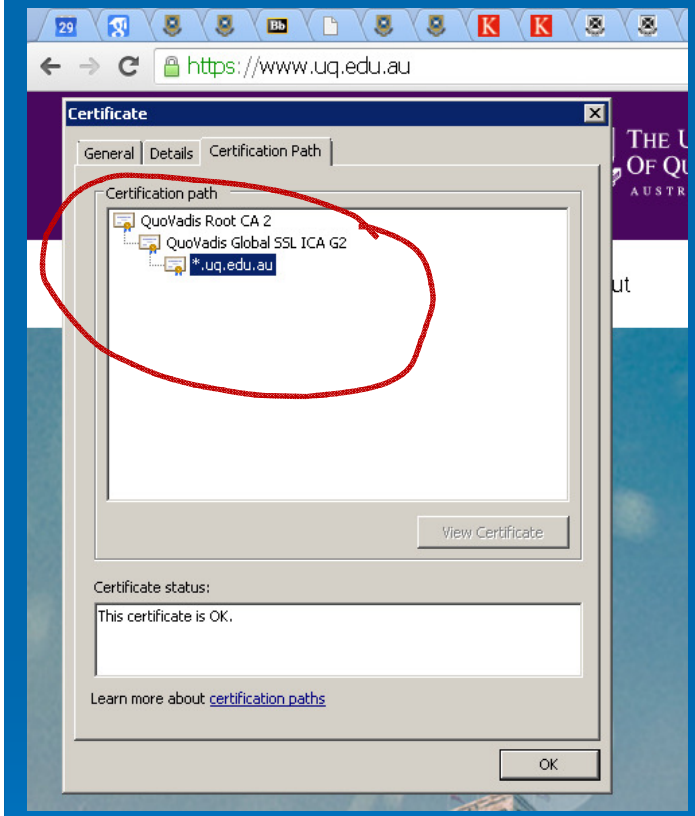
- Structure of a X.509 v3 certificate
  - Certificate
    - Version
    - Serial Number
    - Algorithm ID
    - Issuer
    - Validity
      - Not Before
      - Not After
    - Subject
    - Subject Public Key Info
      - Public Key Algorithm
      - Subject Public Key (e.g. *n* and *e* for RSA)
    - Issuer Unique Identifier (Optional)
    - Subject Unique Identifier (Optional)
    - Extensions (Optional)
      - …
  - Certificate Signature Algorithm
  - Certificate Signature

Certificate Example

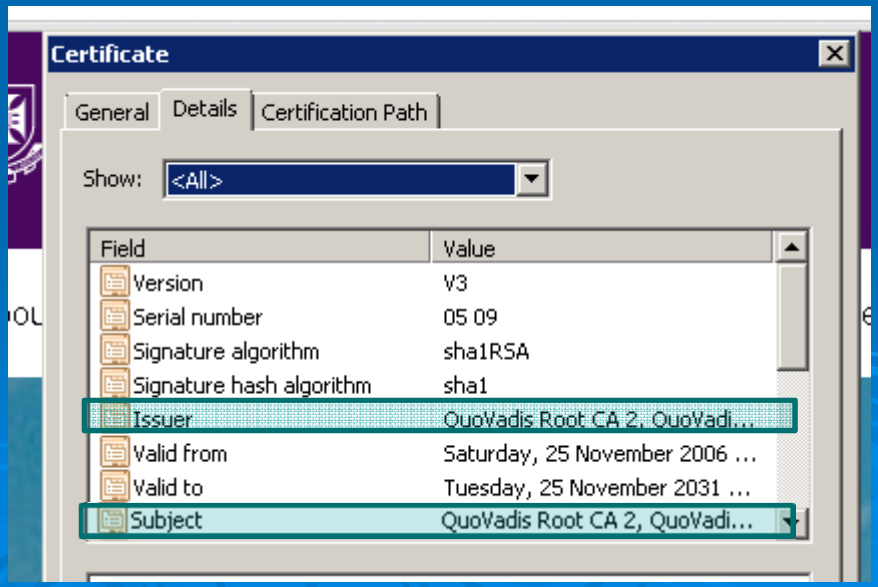


- Certificates are used to establish secure connections in TLS/SSL or HTTPS
  - Discussed later today
- Certification Path shows certification chain

Certificate Chains or Paths



- QuoVadis Root CA 2 is 'Root Certificate' and is 'self-signed'
  - There is no one else to vouch for the authenticity of that public key
  - ~ 100 Root Certificates are pre-installed and implicitly trusted



How to get a Public Key Certificate

- Buy one from a commercial CA
- Make your own self-signed certificate

Certification Authorities (CA)

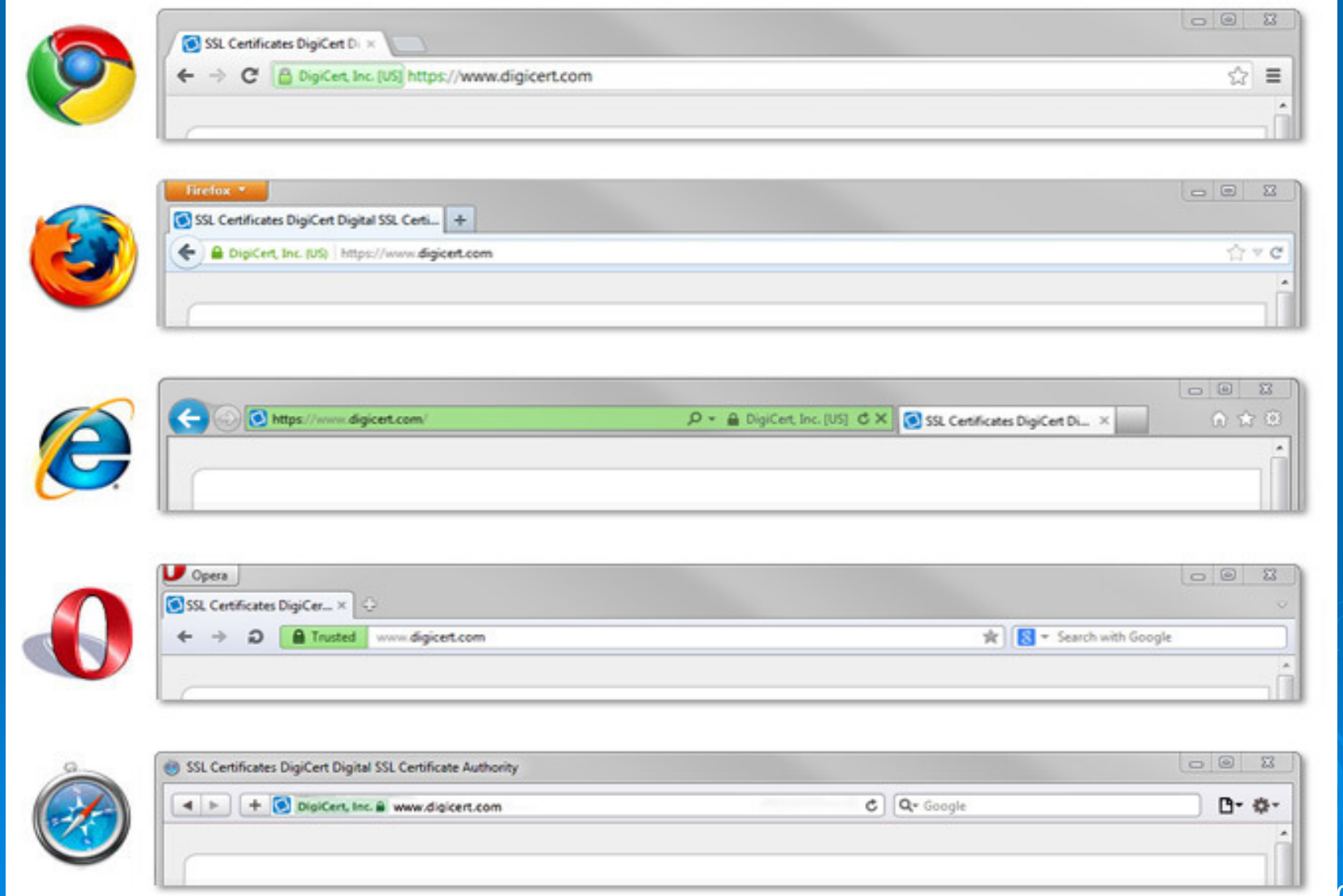
- Entities who sign public key certificates are called
  - **Certification Authorities** (CA)
- Key Commercial CAs

- Organisations/Governments can implement their own, in-house CAs

What do CAs do?

- CAs typically publish details of the identity verification process in a document called the **Certification Practice Statement** (CPS).
  - Example: https://www.symantec.com/content/en/us/about/media/repository/stn-cps.pdf
- CAs typically have different levels of (identity) checking
  - Some merely involve answering an email from a certain address
  - Others may involve presenting at an office with passport etc.
- Extended Validation (EV) Certificate
  - More rigorous verification and validation of application by CA
  - More expensive
  - Indicated via green address bar in browser
  - Example:
    - https://www.symantec.com/en/au/page.jsp?id=compare-ssl-certificates
- All this has nothing to do with cryptography:
  - it concerns **trust into the identity-key binding**.

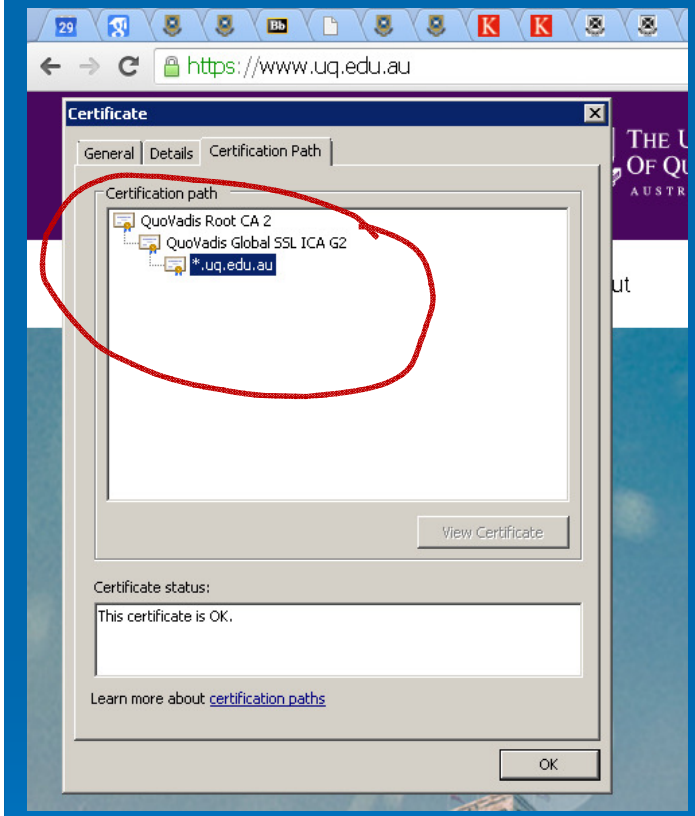
Visual Indication of EV SSL Certificate



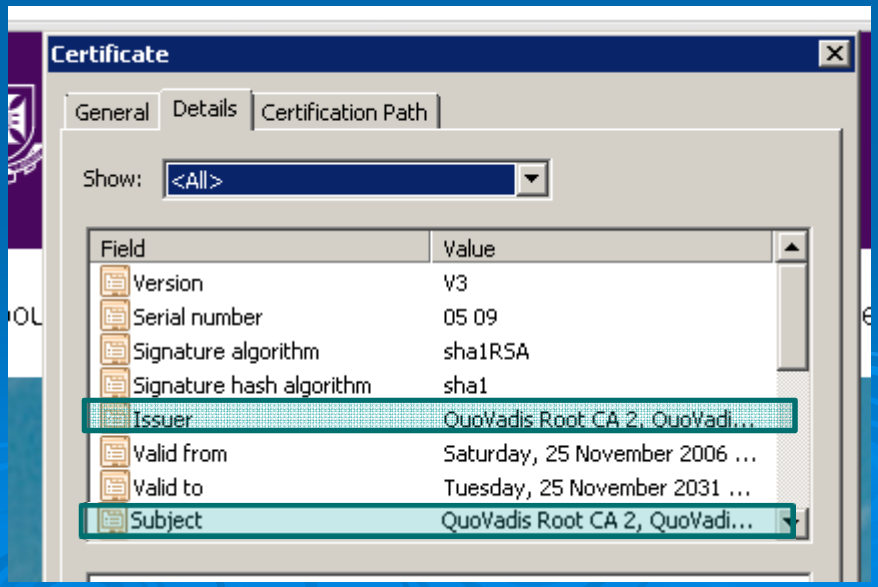
How to create your own self-signed certificate

- Use OpenSSL (http://www.openssl.org/)
  - Open source implementation of SSL/TLS
    - Last year in the news due to Heartbleed vulnerability
  - Also provides generic crypto library
  - Installed on most Linux/Unix systems, also Windows
- What do you need first?
  - First, you need a public key, private key pair
  - `openssl genrsa -des3 -out privkey.pem 2048`
  - http://www.openssl.org/docs/HOWTO/keys.txt
- Now generate self-signed certificate
  - `openssl req -new -x509 -key privkey.pem -out cacert.der -days 10`
  - http://www.openssl.org/docs/HOWTO/certificates.txt
- Display contents of certificate
  - `openssl x509 -in cacert.der -noout -text`
- Now you can import the certificate in a Windows Certificate Store
- You can also generate a Certificate Signing Request (CSR), which you can send to a CA for signing
  - `openssl req -new -key privkey.pem -out cert.csr`

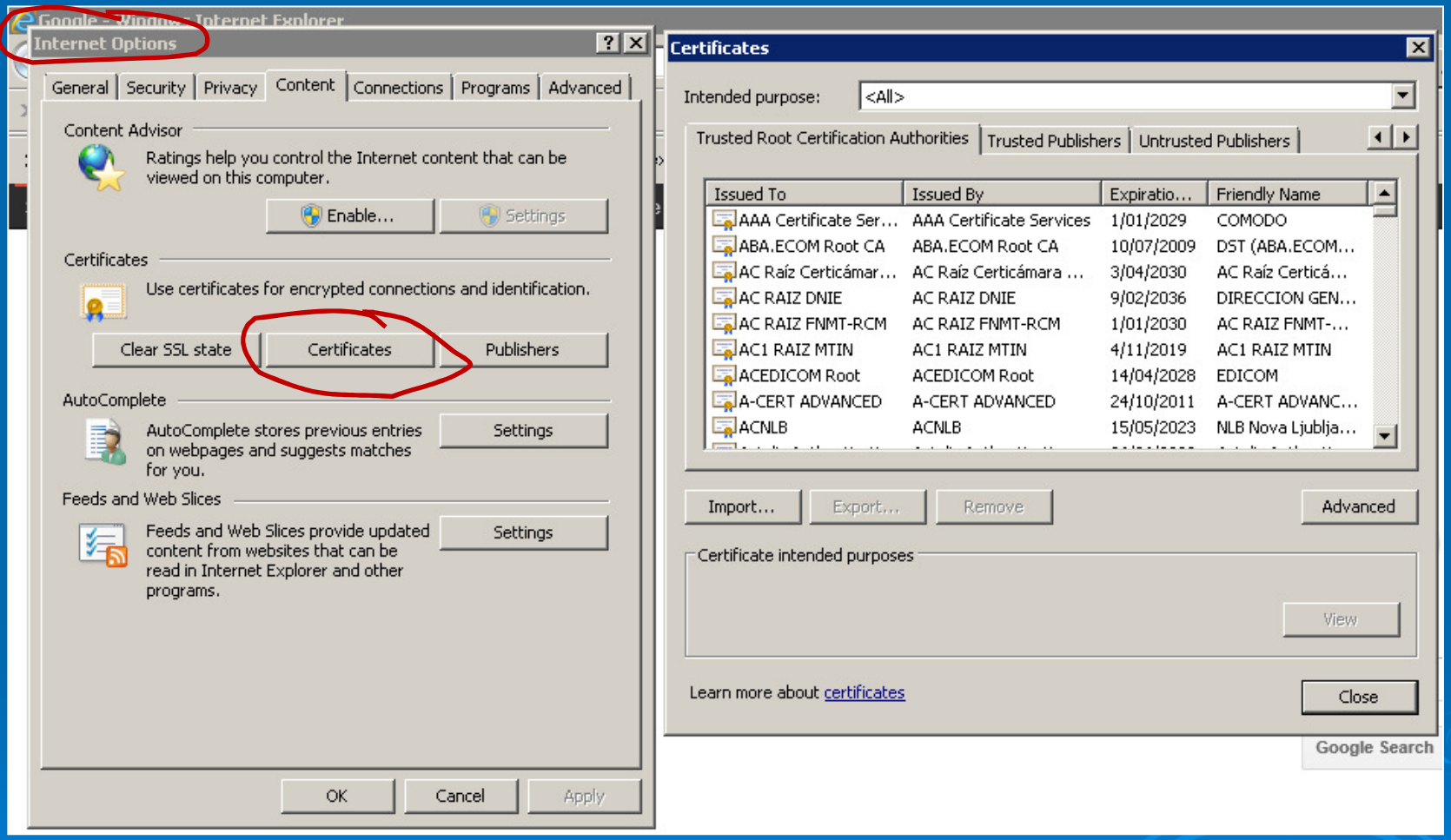
Certificate Chains or Paths



- QuoVadis Root CA 2 is 'Root Certificate' and is 'self-signed'
  - There is no one else to vouch for the authenticity of that public key
  - ~ 100 Root Certificates are pre-installed and implicitly trusted



Root Certificates



- You can add (import) additional Root Certificates

Untrusted Certificates

- What if the certificate or certificate chain cannot be validated by the browser, e.g. if there is no trusted Root Certificate?
  - For example, if the browser receives a self-signed certificate that is not from a trusted Root CA
  - Danger: Could be Man-in-the-Middle attack
- Chrome warning: