Distributed Systems SDEV4001:11
Security in Distributed Systems

December 15st 2010

The Security Problem

- Restrict access to information and resources to just those persons/processes that are authorised to have access
- Broad classes of computer security threats:
  - Data Leakage – loosing control and governance of confidential information
  - Tampering – more than just getting access to bank accounts, consider voter fraud, guarda records, etc
  - Vandalism – making a website more 'attractive'

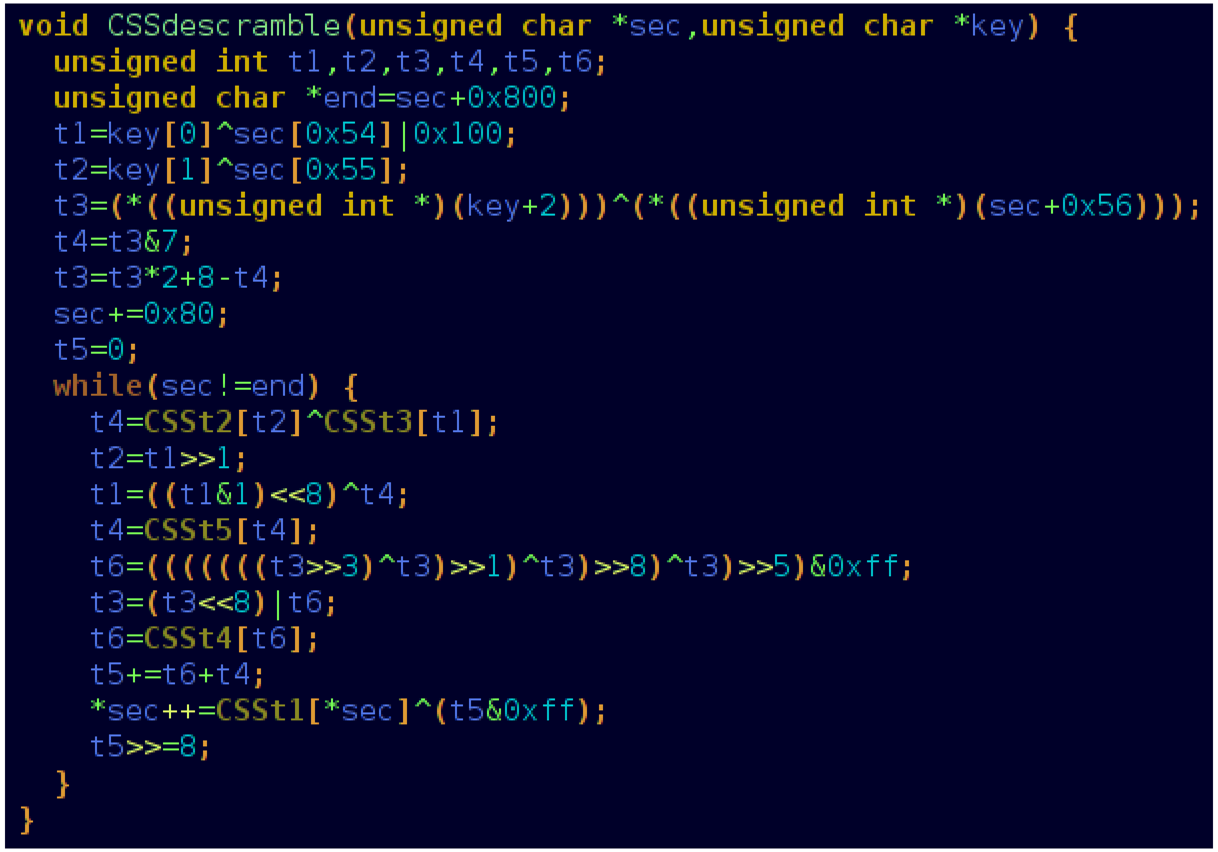
The Security Problem

- Communications channel particularly susceptible to attack:
  - Eavesdropping (interception) – listening in to 'conversations'
  - Masquerading (identity substitutuion) – pretending to be a ligitimate partner
  - Message Tampering – man in the middle
  - Replaying – using eavesdropped messages to initiate a session, etc.
  - Denial of Service – The attacks on Visa etc. Last week

Security Assumptions

- Interfaces are exposed: Hackers can send a message to any communicating process
- Networks are insecure: Addresses can be spoofed
- Algorithms are available to hackers: Secrecy dependent on secrecy of key rather than algorithm
- Attackers have access to large resources: computational power will not be a problem for hackers during the lifetime of the system

- Naive security – DVD security methods were easily cracke through both reverse engineering and brute force methods.



```
void CSSdescramble(unsigned char *sec,unsigned char *key) {
  unsigned int t1,t2,t3,t4,t5,t6;
  unsigned char *end=sec+0x800;
  t1=key[0]^sec[0x54]|0x100;
  t2=key[1]^sec[0x55];
  t3=(*((unsigned int *)(key+2)))^(*((unsigned int *)(sec+0x56)));
  t4=t3&7;
  t3=t3*2+8-t4;
  sec+=0x80;
  t5=0;
  while(sec!=end) {
    t4=CSSt2[t2]^CSSt3[t1];
    t2=t1>>1;
    t1=((t1&1)<<8)^t4;
    t4=CSSt5[t4];
    t6=(((((((t3>>3)^t3)>>1)^t3)>>8)^t3)>>5)&0xff;
    t3=(t3<<8)|t6;
    t6=CSSt4[t6];
    t5+=t6+t4;
    *sec++=CSSt1[*sec]^(t5&0xff);
    t5>>=8;
  }
}
```

A portion of DeCSS

- CSS Key, 40 bits in principle but in practice closer to 16!!!

Cryptography

- 2 classes
  - Symmetric (shared secret key)
  - Asymmetric (public / private key pair)
- 2 roles:
  - Secrecy and integrity
  - Authentication
- Consider
  - Scenarios
  - Algorithms

Cryptographic Notation

K_A	Alice's secret key
K_B	Bob's secret key
K_{AB}	Secret key shared between Alice and Bob
K_{Apriv}	Alice's private key (known only to Alice)
K_{Apub}	Alice's public key (published by Alice for all to read)
$\{M\}_K$	Message M encrypted with key K
$[M]_{\mathrm{K}}$	Message M signed with key K

Cryptographic Examples

- Explain basic usage scenarios

Scenario 1 – Simple Shared Keys

- Alice wishes to send some secret information to Bob. Alice and Bob share a secret key K_{AB}.
  - Alice encrypts M using agreed encryption algorithm $E(K_{AB}, M)$ and shared key K_{AB} producing $\{M_i\}_{K_{AB}}$
  - Bob decrypts with decryption algorithm $D(K_{AB}, M)$
- Problems:
  - How do Alice and Bob exchange keys?
  - How does Bob know that $\{M_i\}$ isn't a replay of an earlier message?

Scenario 2 – Authentication Server

- Alice wishes to access files stored by Bob (a file server).
- Sara is a securely managed authentication server.
- Sara issues users with passwords and holds secret keys for all the users in the system.
  - Sara knows Alice's key K_A and Bob's key K_B.
- Process is described on the next slide.

Scenario 2 – Authentication Server

- Alice sends a message to Sara requesting access to Bob
- Sara sends a message to Alice encrypted with K_A consisting of a Ticket and a new secret key for communication with Bob – K_{AB}
  - The full message is $\{\{Ticket\}K_B, K_{AB}\}K_A$
  - The encrypted ticket contains the identity of Alice and the shared key K_{AB} i.e. $\{Alice, K_{AB}\}$
- Alice decrypts the response giving her $\{Ticket\}K_B$ and K_{AB}
- Alice sends a request R to Bob. The request is $\{Ticket\}K_B$, *Alice*, R
- Bob decrypts the ticket, giving him Alice's identity and the shared secret key. K_{AB} is now used by Alice and Bob for the duration of the session.

Scenario 3 – Using Public Keys

- Bob generates a public/private key pair
  - Asymmetric set of keys where one decrypts a message encrypted by the other
  - Keys are $K_{B_{pub}}$ and $K_{B_{priv}}$
- Alice wants to communicate with Bob using a shared secret key K_{AB}.
- Process is described on next slide

Scenario 3 – Using Public Keys

- Alice accesses a key distribution service to obtain a public key certificate giving Bob's public key ($K_{B_{pub}}$) – the certificate is signed by a trusted authority – a reliably and well known third party
- Alice creates a new secret key K_{AB} and encrypts it with $K_{B_{pub}}$ and sends a message to Bob – including an identifier for the key used to encrypt – in case Bob had more than one
  - *keyname*, $\{K_{AB}\}_{K_{B_{pub}}}$
- Bob selects $K_{B_{priv}}$ from his local keystore and decrypts K_{AB}

Scenario 4 – Document Signing

- Alice wants to sign a document M so that any subsequent recipient can verify that she is the originator of it
- Process:
  - Alice computes a fixed length digest of the document *Digest(M)*
  - Alice encrypts the digest using her private key and appends it to M giving
    - $M, \{Digest(M)\}_{K_{A_{priv}}}$
  - Bob receives the message, extracts M and computes Digest(M)
  - Bob decrypts $\{Digest(M)\}_{K_{A_{priv}}}$ using $K_{A_{pub}}$ and compares the result with his computed Digest(M)
- $M, \{Digest(M)\}_{K_{A_{priv}}}$ represents a digital signature

Cryptographic Algorithms

- Symmetric Encryption
  - Encryption Function
  - Decryption Function
  - Shared Secret Key
- Asymmetric Encryption
  - Encryption Function
  - Decryption Function
  - Encryption Key
  - Decryption Key

Symmetric Cryptographic Algorithms

- $E(K,M) = \{M\}_K$
  - Remember K = key, M = Message, $\{M\}_K$ message encrypted with K
- Basic principles
  - Confusion: Non-destructive operations such as XOR and circular shifting are used to combine blocks of data with the key
  - Diffusion: There is usually repetition and redundancy in plaintext. Regular patterns should be dissipated to avoid frequency analysis

Tiny Encryption Algorithm (TEA)

- Simple and effective
- Uses rounds of integer addition, XOR and bitwise shifts
    - These are operations from different types of math operators – improves effectiveness
- Plaintext is 64-bit block. Key is 128-bit.
- On each of 32 rounds, 2 halves of text are repeatedly combined with shifted portions of key and eachother
- Decryption function is the inverse
- 128 bit key is secure against brute force attack

Tiny Encryption Algorithm (TEA)

```
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];                         1
    unsigned long delta = 0x9e3779b9, sum = 0; int n;               2
    for (n= 0; n < 32; n++) {                                       3
        sum += delta;                                               4
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);       5
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);       6
    }
    text[0] = y;  text[1] = z;                                      7
}
```

Tiny Encryption Algorithm (TEA)

```
void decrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = delta << 5;  int n;
    for (n= 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
        y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
        sum -= delta;
    }
    text[0] = y; text[1] = z;
}
```

Tiny Encryption Algorithm (TEA)

```
void tea(char mode, FILE *infile, FILE *outfile, unsigned long k[]) {
/* mode is ’e’ for encrypt, ’d’ for decrypt, k[] is the key.*/
    char ch, Text[8]; int i;
    while(!feof(infile)) {
        i = fread(Text, 1, 8, infile);          /* read 8 bytes from infile into Text */
        if (i <= 0) break;
        while (i < 8) { Text[i++] = ' ';}       /* pad last block with spaces */
        switch (mode) {
        case 'e':
            encrypt(k, (unsigned long*) Text); break;
        case 'd':
            decrypt(k, (unsigned long*) Text); break;
        }
        fwrite(Text, 1, 8, outfile);            /* write 8 bytes from Text to outfile */
    }
}
```

Data Encryption Standard (DES)

- 56-bit key.
- Handles 64-bit blocks.
- 16 key dependent stages – rounds.
- Data is bit rotated by an amount determined by the key, and 3 key independent transpositions are carried out.
- Cracked in 1997 in 12 weeks
- In 1998 a machine was built to crack DES in 3 days
- Triple DES applied DES three times with 2 keys – 112 bit key.

Other Symmetric Algorithms

- Advanced Encryption Standard
    - “Rijndael” cypher
    - Another example of a block cypher
    - 128 bit block size
    - Keys up to 256 bit
- RC4
    - Ron Rivest
    - Keys up to 256 bit
    - Easy to implement
    - Much faster than AES but less secure
    - Stream cypher
    - Used in WiFi networks

Asymmetric Algorithms

- Avoiding problem of sharing a private key
- Basic Principles
  - Keys K_e and K_d – both very large numbers
  - Encryption function performs exponentiation or some other mathematical function on M using K_e.
  - Decryption performs similar function on $\{M\}_{K_e}$ using K_e.

RSA

- Created by Rivest, Shamir, Adelman
- Suitable for encryption and document signing
- Based on the product of 2 very large prime numbers – both greater than 10^{100}
- Determining prime factors of such large composite numbers is computationally (almost) impossible
- Three steps: key generation, encryption and decryption
- Key generation algorithm is given on next slide

RSA

- We need 6 numbers, P,Q,N,Z,d,e
- Choose two large prime numbers, P and Q and calculate

  N = P * Q

  Z = (P-1) * (Q-1)
- Choose any number that is coprime with Z i.e. It and Z have no common factors bar 1
  - This is 'd'
- E.g. (with small numbers!!!!!)

  P = 13, Q = 17, N = 221, Z = 192, d = 5

RSA

- To find e solve

  $e * d = 1 \bmod Z$

- E.g.

  $e*d = 1 \bmod 192$

  $= 1, 193, 385…$

  385 divisible by *d*

  E = 385 / 5 = 77

N, e, and d are the basis for the actual encryption / decryption process

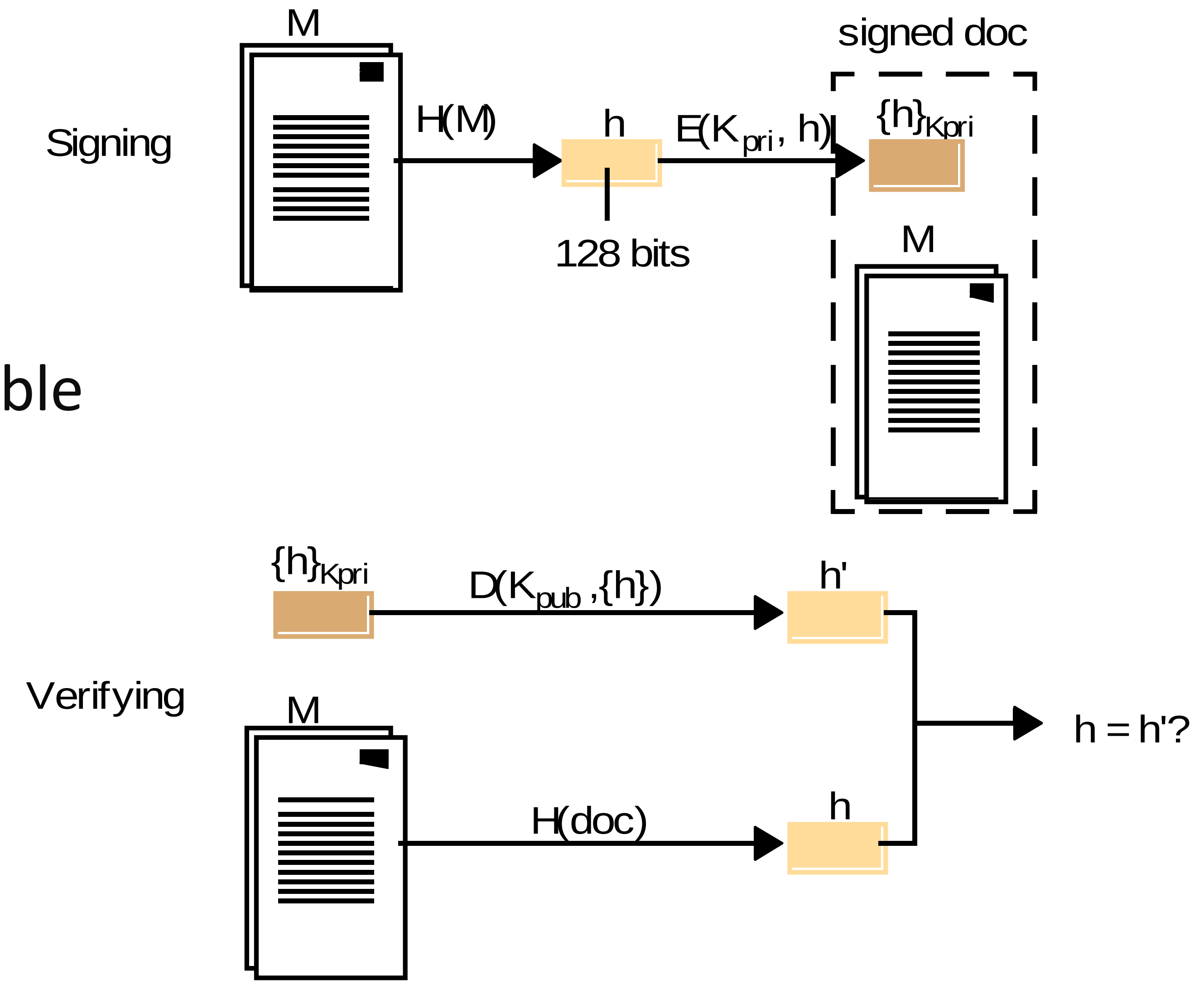
RSA

- To encrypt using RSA, divide plaintext into equal blocks of length *k* bits, where $2^k < N$
- To encrypt a block of plaintext M,
  - $E(e, N, M) = M^e \bmod N$
- E.g.

  $\{M\} = M^{77} \bmod 221$
- To decrypt a block of ciphertext c,
  - $D(d, N, c) = c^d \bmod N$

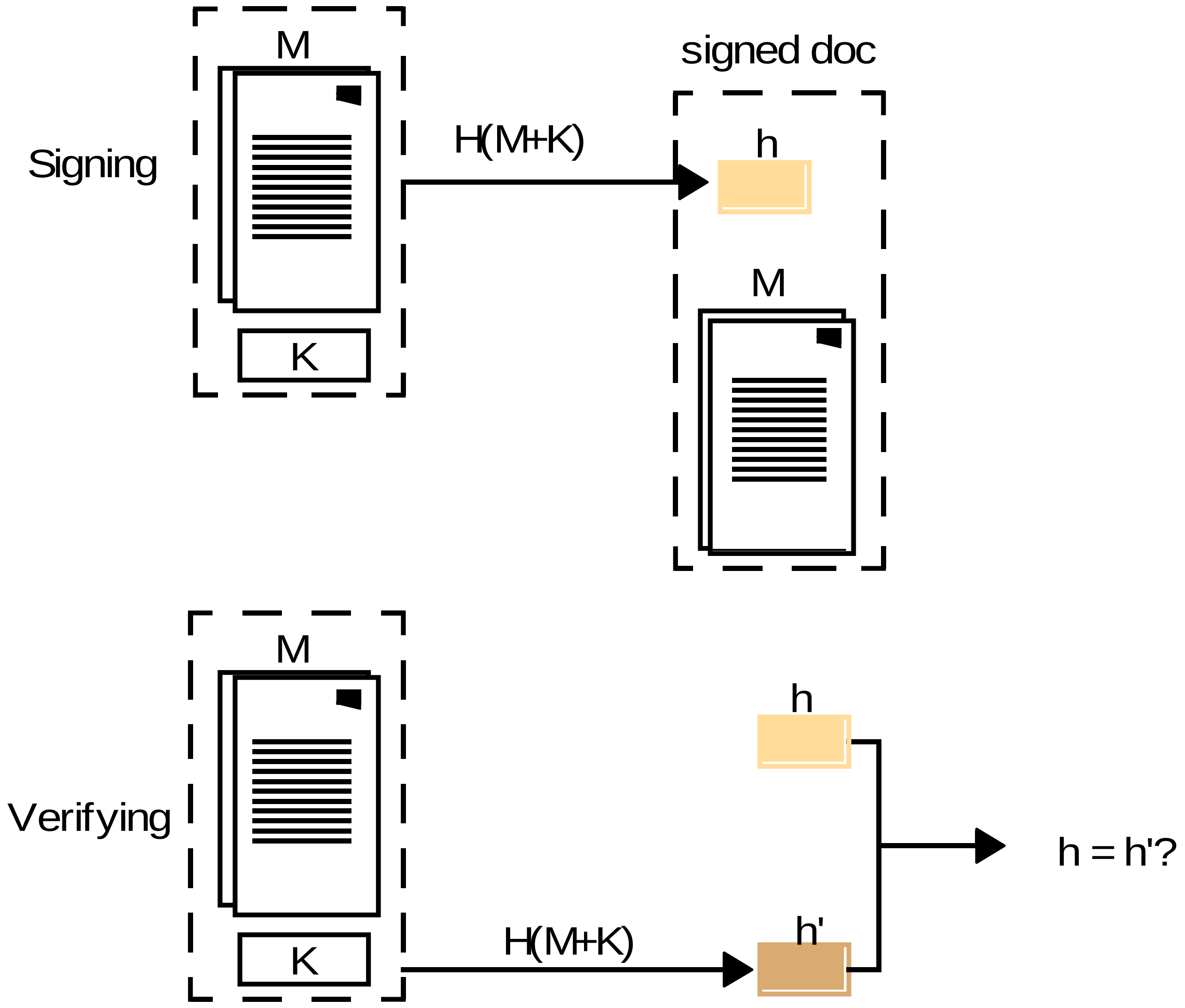
Digital Signatures with Pub/Priv Keys

- Requirements
  - Authentic
  - Unforgeable
  - Non-repudiable



Remember Scenario 4 earlier!!!

Digital Signatures with Shared Secret Keys

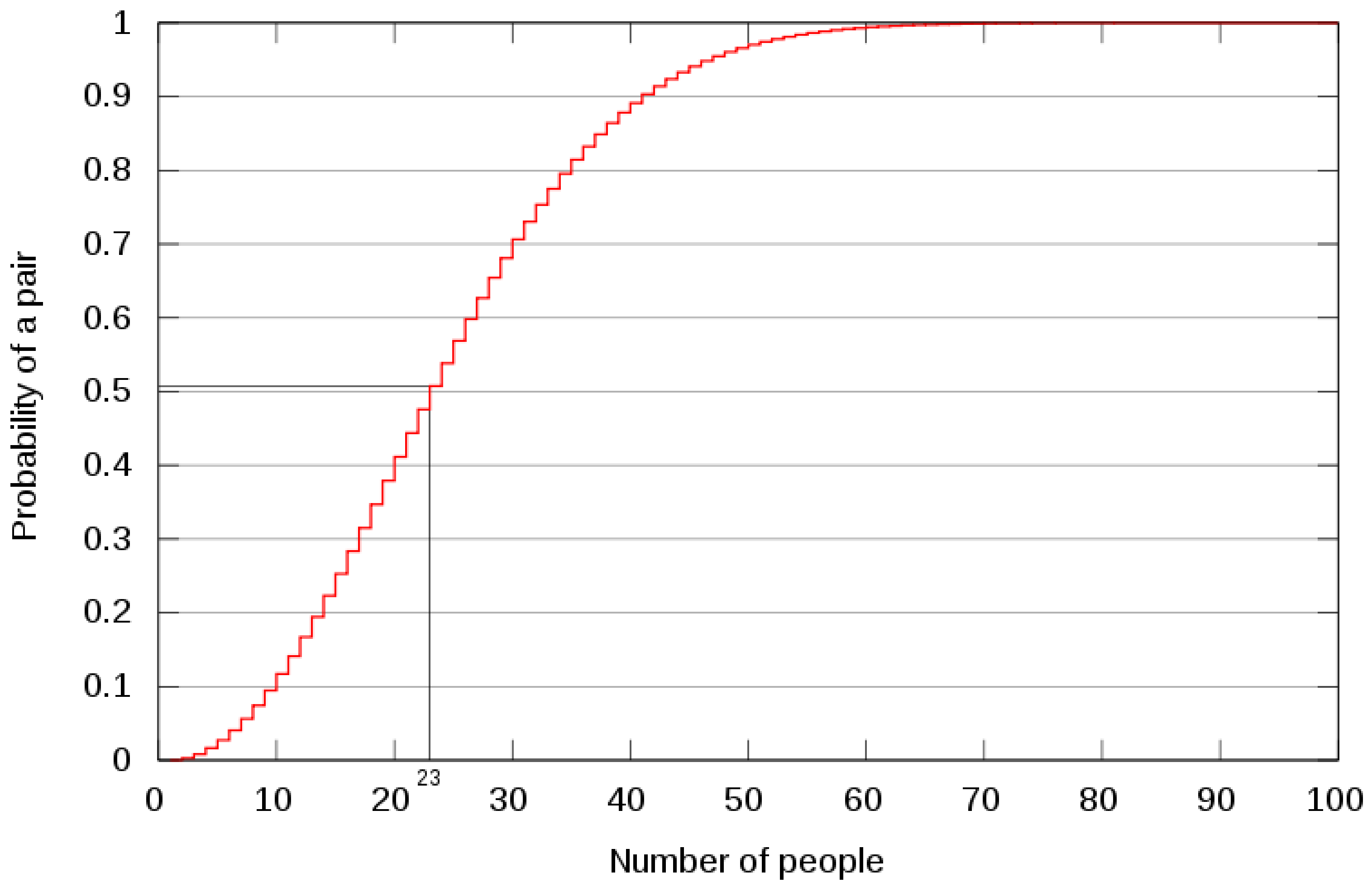


Secure Digest Functions / Hashcodes

- h = H(M)
- Required properties
  - Given M h is easy to compute
  - Given h, M is hard to compute
  - Given M, it is hard to compute another message M' such that H(M) = H(M')
- Hash function should be one way
- Also used in password verification
  - We store the hash of the password, not the password

Birthday Problem

Probabilities of having a pair of birthdays on the same day in the group



Birthday Attack

- Alice prepares two versions of a contract for Bob: M which is favourable to Bob and M’ which is not
- Alice makes many subtly difference versions of M and M’ (adding whitespace etc.) and computes h for each, until h for both is equal
- Alice gets Bob to sign M and then copies the signature to M’

Birthday Attack

- If h is 64 bits long, then we only require 2^{32} versions of M and M’ on average
- We need to use 128 bit h values to guard against birthday attacks
- We use the same operations that are used in symmetric cryptography, although they no longer need to be reversible
- Common algorithms
  - MD5: 4 rounds of 4 functions on 16 32-bit segments of 512-bit blocks to produce 128-bit digest
  - SHA-1: 160 bit digest

Case Study: Java

- Some Java code is automatically downloaded across the network and runs on your machine
- This makes it very important to limit the sorts of things that Web-based Java programs can do
- Simply put, a hostile Java program could trash your machine

Case Study: Java

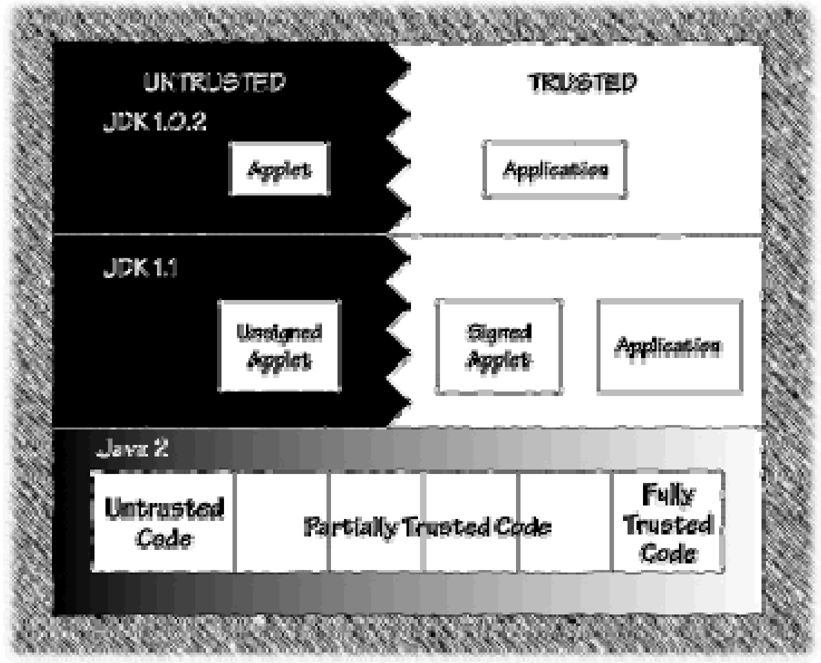
- Java security is essential to developers as well. As a platform, Java has much to offer in terms of security:
  - Java has advanced cryptography Application Program Interfaces (APIs) and class libraries.
  - Java includes language-level security mechanisms that can help make developing secure code easier
  - Some aspects of Java make it more difficult to write insecure (unsafe) code

Case Study: Java

Threat	Explanation & consequence	Java Defence
System Modification	The most severe class of attacks. Applets that implement such attacks are attack applets. Consequences of these attacks: severe.	Strong
Invasion of Privacy	If you value your privacy, this attack class may be particularly odious. They are implemented by malicious applets. Include mail forging. Consequences of these attacks: moderate.	Strong
Denial of Service	Also serious but not severely so, these attacks can bring a machine to a standstill. Also implemented by malicious applets. May require reboot. Consequences of these attacks: moderate.	Weak
Antagonism	Merely annoying, this attack class is the most commonly encountered. Implemented by malicious applets. May require restart of browser. Consequences of these attacks: light to moderate.	Weak

Trust

- **Trusted -v- Untrusted code** has been replaced by levels of trust
- Trusted code can do anything, untrusted code is limited
- With the introduction of Java 2, Java includes the ability to create and manage security policies that treat programs according to their trust level



What untrusted code can’t do (I)

1. Read files on the client file system.
2. Write files to the client file system.
3. Delete files on the client file system, using the File.delete() method
4. Rename files on the client file system, using the File.renameTo() method
5. Create a directory on the client file system, using the File.mkdirs()
6. List the contents of a directory
7. Check to see whether a file exists.
8. Obtain information about a file, including size, type, and modification timestamp.

What untrusted code can’t do (II)

9. Create a network connection to any computer other than the host from which it originated
10. Listen for or accept network connections on any port on the client system
11. Create a top-level window without an untrusted window banner
12. Obtain the user's username or home directory name through any means, including trying to read the system properties: user.name, user.home, user.dir, java.home, and java.classpath.
13. Define any system properties
14. Run any program on the client system using the Runtime.exec() methods

What untrusted code can't do (III)

15. Make the Java interpreter exit, using either System.exit() or Runtime.exit().
16. Load dynamic libraries on the client system using the load() or loadLibrary() methods of the Runtime or System classes.
17. Create or manipulate any thread that is not part of the same ThreadGroup as the applet.
18. Create a ClassLoader
19. Create a SecurityManager
20. Specify any network control functions, including ContentHandlerFactory, SocketImplFactory, or URLStreamHandlerFactory.
21. Define classes that are part of packages on the client system

Java Sandbox

- The Verifier helps ensure type safety
  - Is an object really of the right class?
- The Class Loader loads and unloads classes dynamically from the Java runtime environment
- The Security Manager acts as a security gatekeeper guarding potentially dangerous functionality

The Verifier

- When Java code arrives at the VM and is formed into a Class by the Class Loader, the Verifier examines it to
  - Make sure that the format of a code fragment is correct
  - Make sure that byte code does not forge pointers, violate access restrictions, or access objects using incorrect type information
- If the Verifier discovers a problem with a class file, it throws an exception, loading ceases, and the class file never executes

The Verifier

- Once byte code passes through verification, the following things are guaranteed:
    1. The class file has the correct format
    2. Stacks will not be overflowed or underflowed
    3. Byte code instructions all have parameters of the correct type
    4. No illegal data conversions (casts) occur
    5. Private, public, protected, and default accesses are legal
    6. All register accesses and stores are valid

The Class Loader

- Every mobile code system requires the ability to load code from outside a system into the system dynamically
- Class loaders determine when and how classes can be added to a running Java environment
- A fake Security Manager must not be allowed to load into the Java environment and replacing the real Security Manager. This is known as class spoofing.

The Class Loader

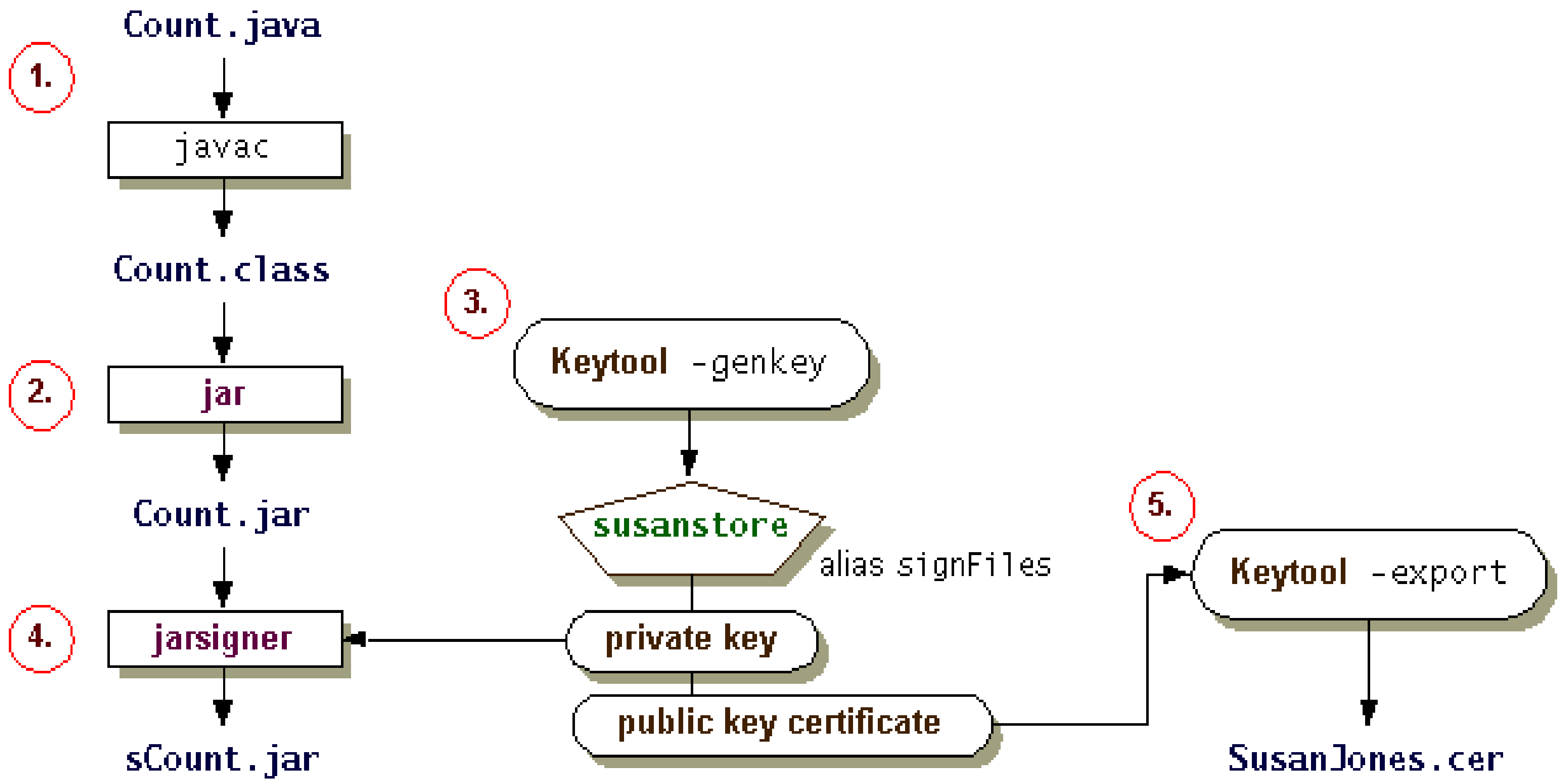
- Class loading proceeds according to the following general algorithm:
  1. Determine whether the class has been loaded before. If so, return the previously loaded class.
  2. Attempt to load the class from the local CLASSPATH. This prevents external classes from spoofing trusted Java classes.
  3. See whether the Class Loader is allowed to create the class being loaded. The Security Manager makes this decision. If not, throw a security exception.
  4. Read the class file into an array of bytes and construct a Class object and its methods from the class file.
  5. Resolve any static classes referenced by the class before it is used.
  6. Check the class file with the Verifier.

The Security Manager

- The job of the Security Manager is to keep track of who is allowed to do which dangerous operations
- A standard Security Manager will disallow most operations when they are requested by untrusted code, and will allow trusted code to do whatever it wants
- Java's Security Manager works as follows:
    1. A Java program makes a call to a potentially dangerous operation in the Java API.
    2. The Java API code asks the Security Manager whether the operation should be allowed
    3. The Security Manager throws a SecurityException if the operation is denied.
-

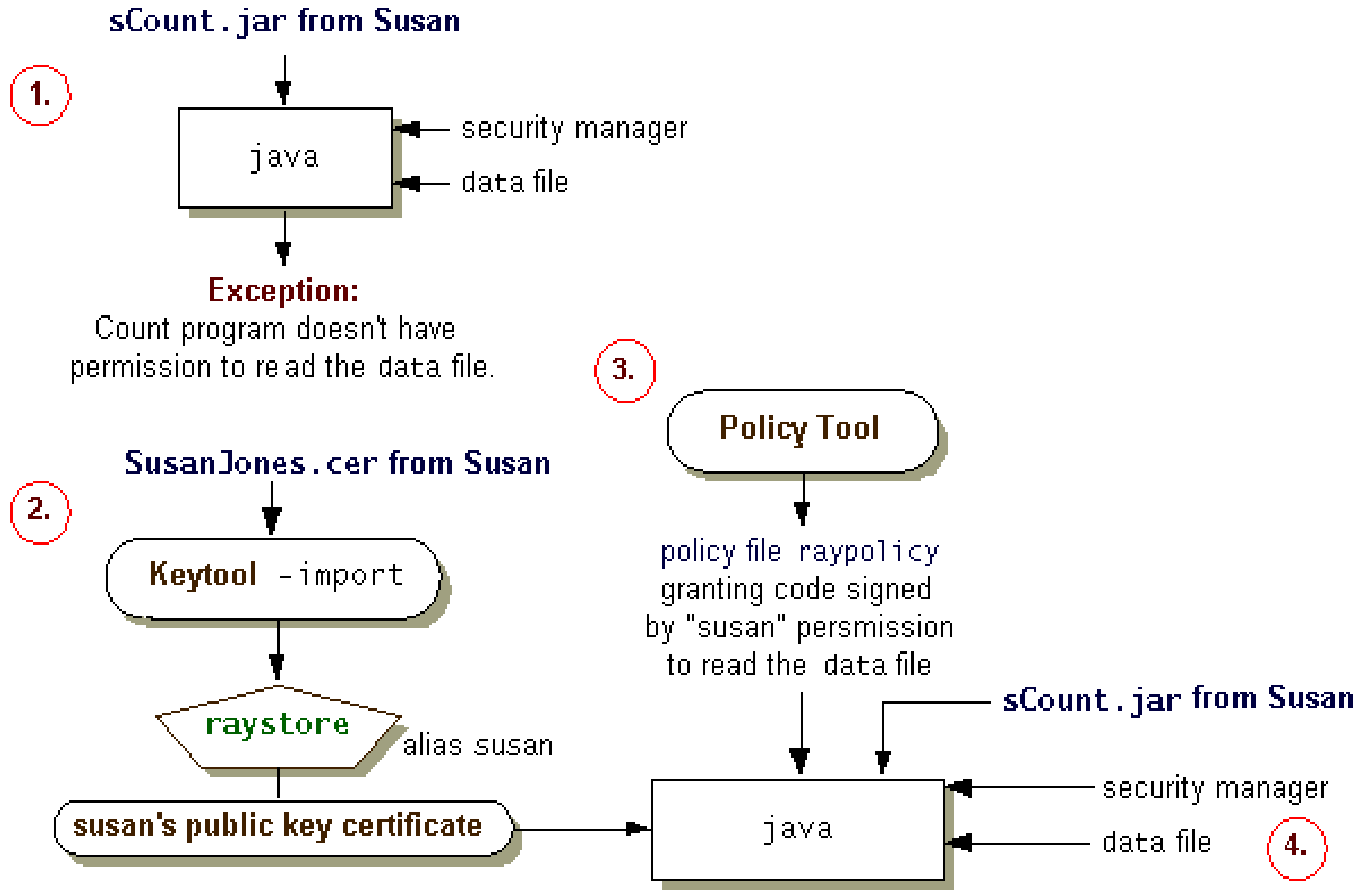
Signing Code and Granting Permissions

- Example: Susan sending to Ray



Signing Code and Granting Permissions

- Steps on receiving side



References

- Distributed Systems, Concepts and Design (4th Edition), by George Coulouris, Jean Dollimore and Tim Kindberg
  - Chapter 7