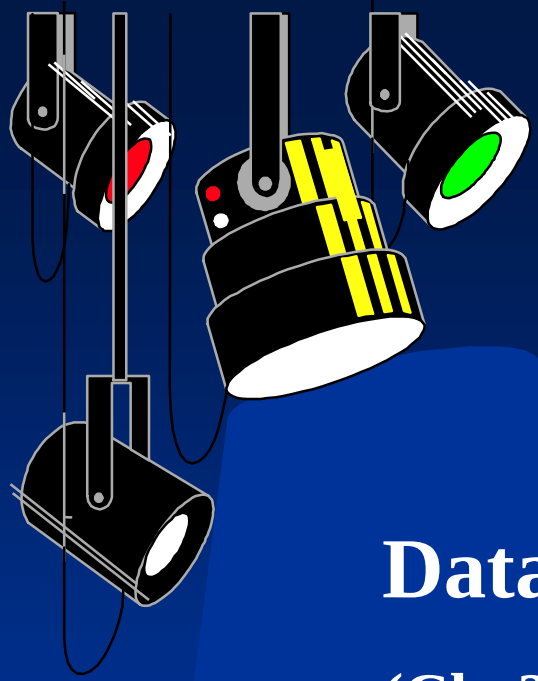


Database security and authorization

(Ch. 22, 3rd ed. – Ch. 23, 4th ed. – Ch. 24, 6th – Ch. 30, 7th ed.)

- **Database Security Control**
- **SQL Injection**
- **Encryption and Decryption**

How do we protect the database from unauthorized access?

Who should be able to see employee salaries, student grades, … ?

Who should be able to update … ?

Techniques include/involve:

- passwords
- log-in process - new entry for the log:
  {who, where, when}
- privileges
- Encryption and decryption
- accounts - system (DBA), user

We study two mechanisms:

- Discretionary access control (DAC)
  - privileges such as read, write, update are granted to users
  - a certain amount of discretion is given to the owner of data or anyone else with appropriate authority
- Mandatory access control (MAC)
  - multilevel security is applied to data and users
  - controlled by a central authority, not by owners of an object
  - the owner/creator of an object does not decide who has clearance to see the object

Note:

- US DoD (US Department of Defense) has four basic divisions for systems: A, B, C, D

- lowest to highest is D, C1, C2, B1, B2, B3, A1

- C1, C2, B1, B2, B3, A1 require DAC
- B1, B2, B3, A1 require MAC

Discretionary access control

(in the context of relational systems)

- based on granting and revoking of privileges
- privileges are assigned at two levels:
  - account (user) - each account can be assigned privileges (rights or capabilities)
  - relation - the privilege to access a particular relation is controlled (restricted)

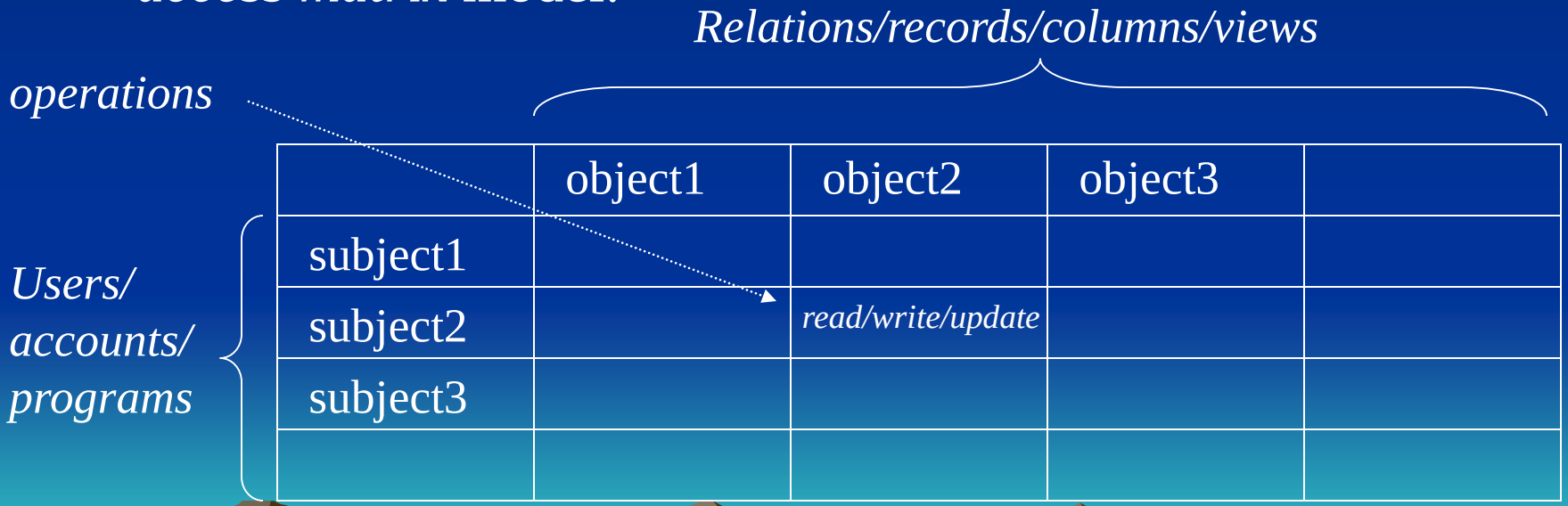
Discretionary access control

- account level
  - create - schema, table, view
  - alter - indexes, table (attributes, indexes)
  - drop - table, index, view
- example
  - grant *createtab* to A1;

  if A1 now creates a table X, then A1 *owns* X, and has all privileges on table X, and A1 can grant privileges to others

Discretionary access control

- relation level
  - privileges on relations and columns
  - type of access: read, write, update
  - *access matrix* model:



Discretionary access control:

suppose A1 executes:

- create table employee (…);
- create table department (…);
- grant insert, delete on employee, department to A2;
- grant select on employee, department to A3;
- grant update on employee(salary) to A4;

	employee	department	employee.salary	
A1	*all*	*all*		
A2	*insert, delete*	*insert, delete*		
A3	*select*	*select*		
A4			*update*	

Discretionary access control: Views

- suppose A1 executes:

```
create view EmployeesInDiv5 as
select name, position, manager
from employee e, department d
where d.div = 5 and e.deptno = d.deptno;

grant select on EmployeesInDiv5 to A4;
```

	employee	department	EmployeesInDiv5
A1	*all*	*all*	*all*
A4			*select*

Mandatory access control for multilevel security

Bell LaPadula model:

- specifies the allowable paths of information flow:

  information with high secret - information with low secret

- set of subjects S, and a set of objects O

S:
- user
- account
- programs

O:
- field
- tuple
- column
- relation
- view

•each s in S and o in O has a fixed security class

class(s) a clearance of s

class(o) a classification level of o

•security classes are ordered by <=

U (Unclassified) <= C (confidential) <=

S (Secret) <= TS (Top Secret)

(public <= sensitive <= top secret)

Mandatory access control for multilevel security

Two properties of the *Bell LaPadula* model:

- Simple Security Property

  a subject s is not allowed read access to an object o unless

  class(s) >= class(o)

  To see something, your clearance must be at least that of what you want

In the military model, the security clearance of someone receiving a piece of information must be at least as high as the classification of the information

Mandatory access control for multilevel security

Second property of the *Bell LaPadula* model:

- Star Property

  a subject s is not allowed write access to an object o unless

  class(s) <= class(o)

  To create/update something, your clearance must be no greater than the object you are creating/updating

In the military model, a person writting some information at one level may pass that information along only to people at levels no lower than the level of the person

Mandatory access control for multilevel security

Implementation of the *Bell LaPadula* model:

- for each original attribute in a relation, add a classification attribute
- add a classification attribute for the tuple (row) - value is maximum of all classifications within the tuple
- these classification attributes are transparent to the user

Mandatory access control for multilevel security

Implementation example: suppose U <= C <= S

Employee relation

Name	Salary	JobPerformance
Smith	40,000	Fair
Brown	80,000	Good

The user view without MAC

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	40,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S

system view with MAC

Mandatory access control for multilevel security

Implementation example:

Stored Rows:

we store only the required tuples that then allow us to materialize tuples for lower levels

For example,

Smith	U	40,000	C	Fair	S	S

allows us to materialize tuples for Classes U, C, and S:

U:

Smith	*null*	*null*

C:

Smith	*40,000*	*null*

S:

Smith	*40,000*	*Fair*

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	40,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S

U:

Name	Salary	JobPerformace
Smith	*null*	*null*

C:

Name	Salary	JobPerformace
Smith	*40,000*	*null*
Brown	*null*	*Good*

S:

Name	Salary	JobPerformance
Smith	*40,000*	*fair*
Brown	*80,000*	*Good*

Mandatory access control for multilevel security

Implementation example:

What does a class C user see if he/she executes

Select * from Employee

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	40,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S

The result is filtered

Name	Salary	JobPerformance
Smith	*40,000*	*null*
Brown	*null*	*Good*

Mandatory access control for multilevel security

Implementation example:

What happens if a class C user executes

Update Employee set Salary=100,000

What does the Star Property say?

Let's assume the result is to be class C.

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	40,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S

Class C attribute is updated

Class S attribute forces polyinstantiation

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	100,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S
Brown	C	100,000	C	Good	C	C

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	100,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S
Brown	C	100,000	C	Good	C	C

U:

Name	Salary	JobPerformance
Smith	*null*	*null*

C:

Name	Salary	JobPerformance
Smith	*100,000*	*null*
Brown	*100,000*	*Good*

S:

Name	Salary	JobPerformance
Smith	*40,000*	*fair*
Brown	*80,000*	*Good*

Mandatory access control for multilevel security

Implementation example:

Stored Rows:

our update example required a row to be *polyinstantiated*

For example, updating the Salary field in

Brown	C	80,000	S	Good	C	S

requires two rows for us to be able to materialize records for classes S, C, and U

Two rows with the same key!

Brown	C	80,000	S	Good	C	S
Brown	C	100,000	C	Good	C	C

Mandatory access control for multilevel security

Implementation example:

What does a class C user see if he/she executes

Select * from Employee

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	100,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S
Brown	C	100,000	C	Good	C	C

The result is filtered

Only the row with TC=C is used

Name	Salary	JobPerformance
Smith	*100,000*	*null*
Brown	*100,000*	*Good*

SQL Injection

SQL injection is a web security vulnerability that allows an attacker

- to interfere with the queries that an application makes to its database.
- to view data that they are not normally able to access. This might include data belonging to other users, or any other data that the application itself is not able to access.
- In many cases, an attacker can modify or delete some data, causing persistent changes to the application's content or behavior.
- In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

SQL Injection

What is the impact of a successful SQL injection attack?

- A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information.
- Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines.
- In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

SQL Injection Examples

What is the impact of a successful SQL injection attack?

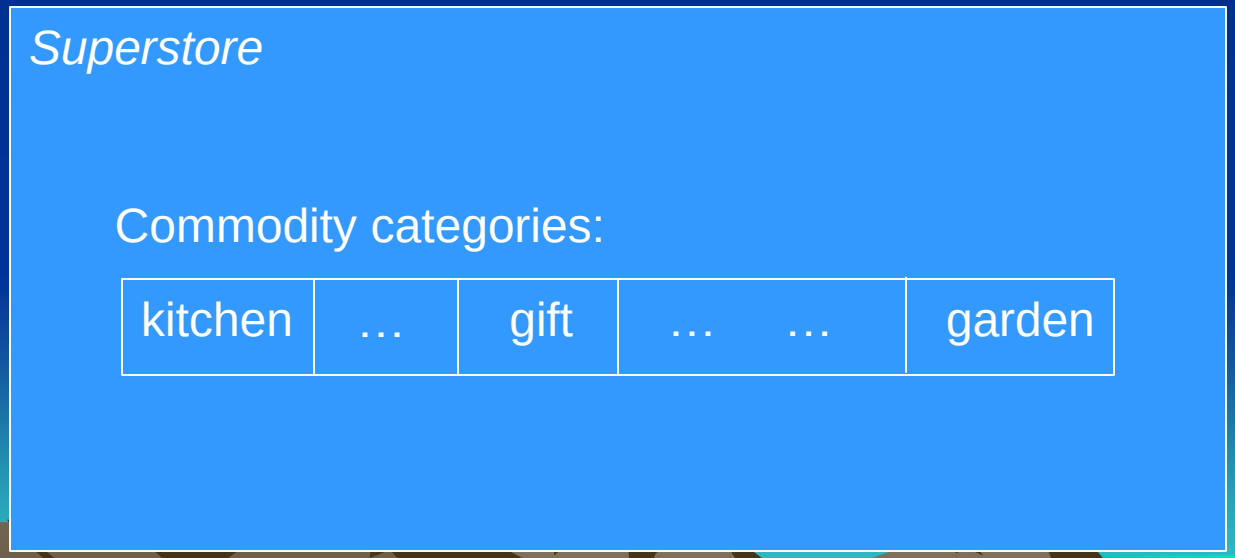
There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

- Retrieving hidden data, where you can modify an SQL query to return additional results.
- Subverting application logic, where you can change a query to interfere with the application's logic.
- UNION attacks, where you can retrieve data from different database tables.
- Examining the database, where you can extract information about the version and structure of the database.
- Blind SQL injection, where the results of a query you control are not returned in the application's responses.

Retrieving hidden data

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser will request an URL.

```
https://insecure-website.com/products?category='Gifts'
```



Retrieving hidden data

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

```
https://insecure-website.com/products?category='Gifts'
```

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products
WHERE category = 'Gifts'
AND released = 1
```

The restriction released = 1 is being used to hide products that are not released. For unreleased products, presumably, released = 0.

This SQL query asks the database to return

- all details (*)
- from the products table
- where the category is Gifts
- and released is 1

The restriction `released = 1` is being used to hide products that are not released. For unreleased products, presumably `released = 0`.

The application doesn't implement any defenses against SQL injection attacks, so an attacker can construct an attack like:

```
https://insecure-website.com/products?category='Gifts'--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'-- AND released = 1
```

The key thing here is that the double-dash sequence -- is a comment indicator in SQL and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes `AND released = 1`. This means that all products are displayed, including unreleased products.

Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about:

```
https://insecure-website.com/products?category='Gifts'+OR+1=1--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1 -- AND released = 1
```

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since 1=1 is always true, the query will return all items.

Subverting application logic

Consider an application that lets users log in with a username and password. If a user submits the username WIENER and the password BLUECHEESE, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
```

- If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

- Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator’ -- and a blank password results in the following query:

```
SELECT * FROM users WHERE username =
  'administrator’-- AND
  password='bluecheese'
```

This query returns the user whose username is administrator and successfully logs the attacker in as that user.

Retrieving data from other database tables

- In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database.
- This is done using the UNION keyword, which lets you execute an additional select query and append the results to the original query.

For example, if an application executes the following query containing the user input "Gifts":

```
SELECT name, description FROM
products WHERE category = 'Gifts'
```

then an attacker can submit the input:

```
 UNION SELECT username, password
FROM users--
```

This will cause the application to return all usernames and passwords along with the names and descriptions of products.

Examining the database

Following initial identification of an SQL injection vulnerability, it is generally useful to obtain some information about the database itself. This information can often pave the way for further exploitation.

You can query the version details for the database. The way that this is done depends on the database type, so you can infer the database type from whichever technique works. For example, on Oracle you can execute:

```
SELECT * FROM v$version
```

You can also determine what database tables exist, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

```
SELECT * FROM information_schema.tables
```

Blind SQL injection vulnerabilities

- Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses.
- Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

Depending on the nature of the vulnerability and the database involved, the following techniques can be used to exploit blind SQL injection vulnerabilities:

- You can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.

- You can conditionally trigger a time delay in the processing of the query, allowing you to infer the truth of the condition based on the time that the application takes to respond.
- You can trigger an out-of-band network interaction, using OAST techniques (Out-of-band application security testing). This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel, for example by placing the data into a DNS-lookup for a domain that you control.

Protection against SQL Injection

Protection against SQL injection attacks can be achieved by applying certain programming rules to all Web accessible procedures and functions.

- **Bind Variables (using parameterized statements)**

  - The use of bind-variables (also known as parameters) protects against injection attacks and also improves performance.

Protection against SQL Injection

Consider the following example using Java and JDBC:

PreparedStatement stmt = conn.preparedStatement(“SELECT * FROM EMPLOYEEE WHERE EMPLOYEE_ID = ? AND PASSWARD = ?”);

stmt.setString(1, employee_id);

stmt.setString(2, passward);

ResultSet resultSet = stmt.executeQuery();

Protection against SQL Injection

Filtering Input (Input Validation)

- This technique can be used to remove escape characters from input strings by using the SQL replace function. For example, the delimiter single quote (’) can be replaced by two single quotes(’’).
- Some SQL manipulation attacks can be prevented by using this technique, since escape characters can be used to inject manipulation attacks.
- However, because there can be a large number of escape characters, this technique is not reliable.

Protection against SQL Injection

Function Security

Database functions, both standard and customed, should be restricted, as they can be exploited in the SQL function injection attacks.

-SQL Aggregate Functions

-SQL Comparison Functions

-SQL String Functions

-SQL Math Functions

-SQL Date Functions

-SQL Window Functions

Need to write a program to restrict these functions.

Encryption and decryption

- Symmetric encryption
- Asymmetric encryption

Encryption and Decryption

Symmetric encryption is a type of encryption where only one key (a secret key) is used to both encrypt and decrypt electronic information.

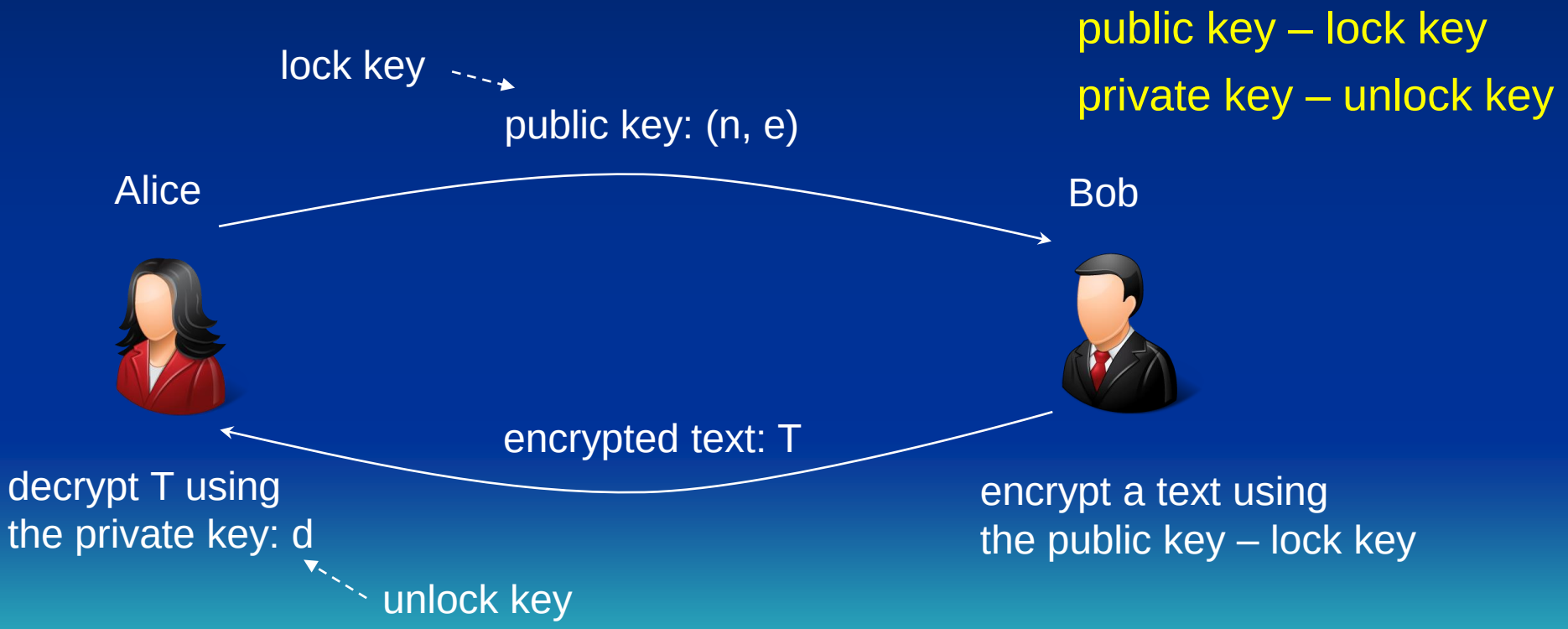
- The entities communicating via symmetric encryption must exchange the key so that it can be used in the decryption process.
- This encryption method differs from asymmetric encryption where a pair of keys, one public and one private, is used to encrypt and decrypt messages.

Encryption and Decryption

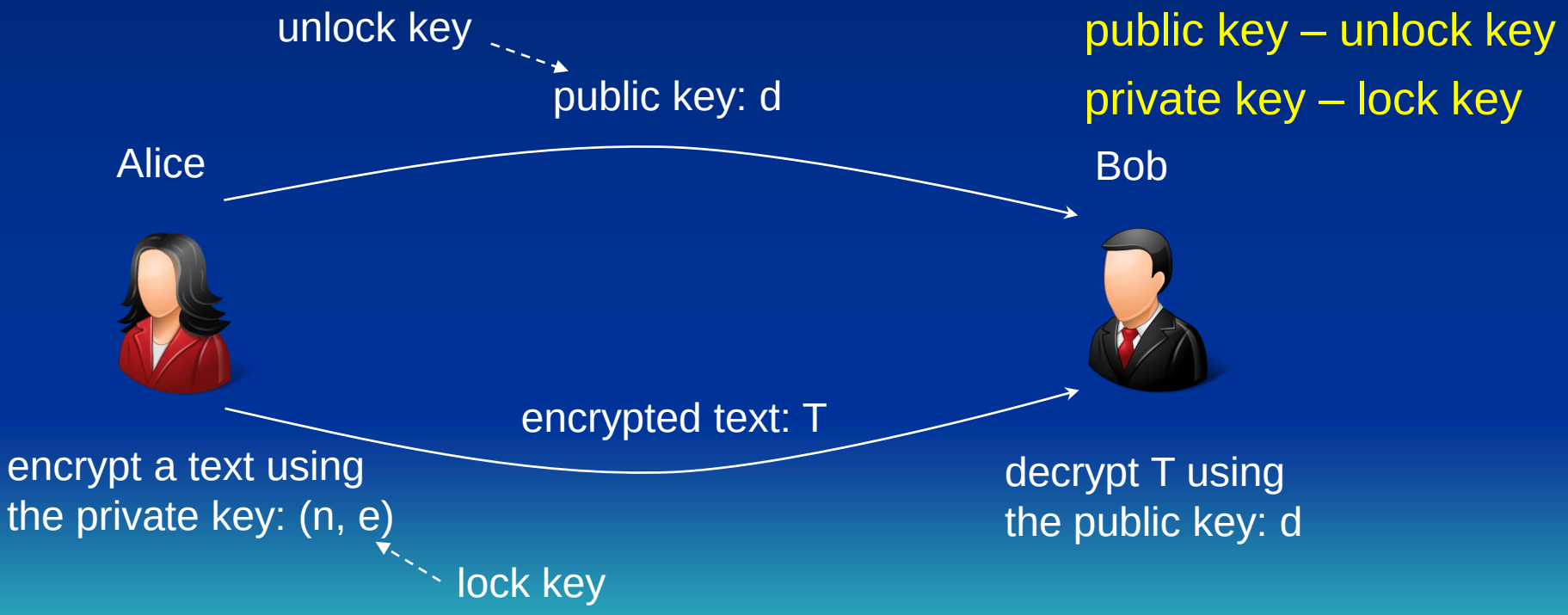
Asymmetric keys are the foundation of Public Key Infrastructure (PKI) - a cryptographic scheme

- Requiring two different keys, one to lock or encrypt the plaintext, and one to unlock or decrypt the cyphertext. Neither key will do both functions.
- One key is published (public key) and the other is kept private (private key).
- This system also is called asymmetric key cryptography.

- If the lock/encryption key is the one published, the system enables private communication from the public to the unlocking key's owner.



- If the unlock/decryption key is the one published, then the system serves as a signature verifier of documents locked by the owner of the private key.



Encryption and Decryption

Key distribution

- Suppose that Bob wants to send information to Alice. If they decide to use *RSA* (an encryption algorithm, proposed by Rivest, Shamir, Adleman), Bob must know Alice's public key to encrypt the message and Alice must use her private key to decrypt the message.
- To enable Bob to send his encrypted messages, Alice transmits her public key (n, e) to Bob via a reliable, but not necessarily secret, route. Alice's private key d is never distributed.

Encryption and Decryption

Encryption

- After Bob obtains Alice's public key (n, e), he can send a message M to Alice.
- To do it, he first turns M (strictly speaking, the un-padded plaintext) into an integer m (strictly speaking, the padded plaintext), such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c, using Alice's public key e, corresponding to

$$m^e \equiv c \pmod{n}$$

This can be done reasonably quickly, even for very large numbers, using modular exponentiation.

- Bob then transmits c to Alice.

Encryption and Decryption

Decryption

•Alice can recover *m* from *c* by using her private key exponent *d* by computing

$$c^d \equiv (m^e)^d \pmod{n}$$

public key= (*n*, *e*)

Given *m*, she can recover the original message *M* by reversing the padding scheme.

How to create a lock (encrypt) key: (*n*, *e*)?

How to create an unlock (decrypt) key: *d*?

How to encrypt integer *m using* (*n*, *e*)?

How to decrypt integer *c* using *d*?

Encryption and Decryption

Example

Determine the public key: (n, e)

1. Choose two distinct prime numbers, such as

$p = 61$, $q = 53$

2. Compute $n = p \times q$ giving

$n = 61 \times 53 = 3233$

3. Compute the Carmichael's totient function of the product as $\lambda(n) = lcm(p - 1, q - 1)$, giving

$\lambda(3233) = lcm(60, 52) = 780$

lcm – least common multiple

4. Choose any number $1 < e < 780$ that is *coprime* to 780.

Choosing a prime number for e leaves us only to check that e is not a divisor of 780.

coprime – two numbers have no common factors other than 1

Encryption and Decryption

Example

4. Choose any number $1 < e < 780$ that is coprime to 780.

Choosing a prime number for e leaves us only to check that e is not a divisor of 780.

Let $e = 17$.

5. Compute d, the modular multiplicative inverse of e (i.e., e is a number satisfying $1 = (e \times d) \bmod 780$), yielding

$d = 413$,

as $1 = (17 \times 413) \bmod 780$.

$(17 \times 413 = 7021 = 9 \times 780 + 1)$

Determine the private key d

Encryption and Decryption

Example

The public key is (n = 3233, e = 17). For a padded plaintext message m, the ciphertext is

$$c = m^e \bmod n = m^{17} \bmod 3233.$$

The private key is d = 413. By using d, we can get

$$m = c^d \bmod n = c^{413} \bmod 3233.$$

For instance, in order to encrypt $m = 65$, we calculate

$$c = 65^{17} \bmod 3233 = 2790$$

To decrypt $c = 2790$, we calculate

$$m = = 2790^d \bmod 3233 = 2790^{413} \bmod 3233 = 65.$$

Encryption and Decryption

Comments:

- Both of these calculations can be computed efficiently using the square-and-multiply algorithm for modular exponentiation.
- In real-life situations the primes selected would be much larger; in our example it would be trivial to factor n, 3233 (obtained from the freely available public key) back to the primes p and q. e, also from the public key, is then inverted to get d (according to $1 = (e \times d) \bmod 780$), thus acquiring the private key.

$n = p \times q = 3233$ $lcm(p - 1, q - 1) = 780$

Modular exponentiation:

$(a \times b) \bmod m$

$= (a \bmod m) \times (b \bmod m) \bmod m$

$2790^{413} \bmod 3233$

$= (2790^{2} \bmod 3233) \times 2790^{411} \bmod 3233$

$= (7784100 \bmod 3233) \times (2790^{411} \bmod 3233)$

$= (249 \bmod 3233) \times (2790^{411} \bmod 3233)$

$= (249 \times 2790 \bmod 3233) \times (2790^{410} \bmod 3233)$

$= \ldots \ldots = 65$