Database Security

IDATG2204 Data Modelling and Database Systems

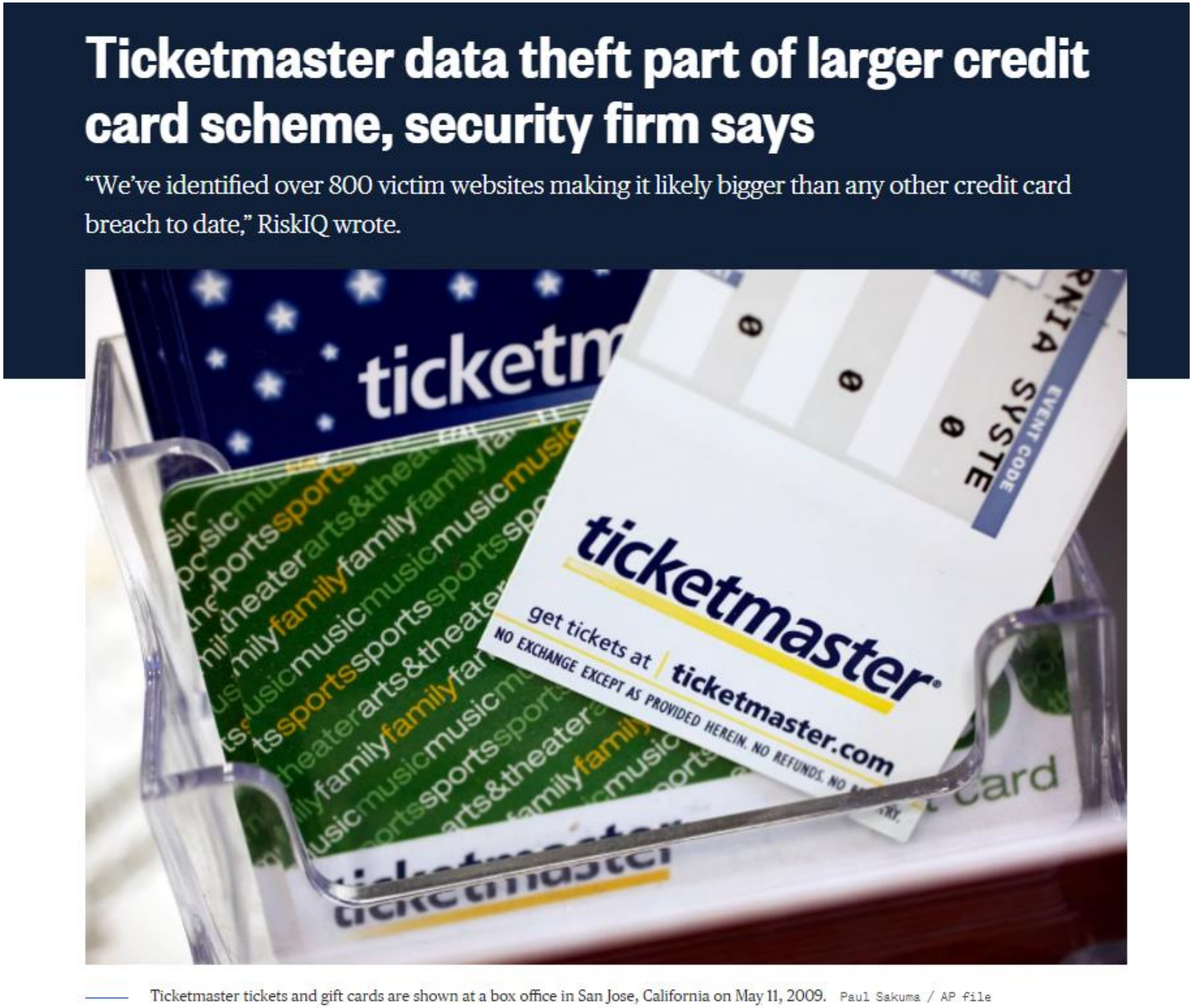
Norwegian University of Science and Technology

Where are We Now?

- W02: Introduction, Relational Algebra
- W03: SQL
- W04: SQL, Conceptual Modelling
- W05: Conceptual Modelling
- W06: Normalisation, Logical Modelling
- W07: Logical Modelling, Physical DB Design, NOSQL
- W08: Physical DB Design, NOSQL
- W09: DB Application Testing, DB Security
- W10-W14: Project Kick-off, Project Work with Peer Review
- W15: Indexing, query processing, concurrency
- W16: Recovery
- W17: More SQL and NOSQL
- W18: Review and Wrap-up

Outline

- Database security:
  - Threats and countermeasures
- Security in database applications:
  - Robust programming
  - Secure HTTP
  - Passwords
  - SQL injection
- Privileges and roles
- Security in MariaDB

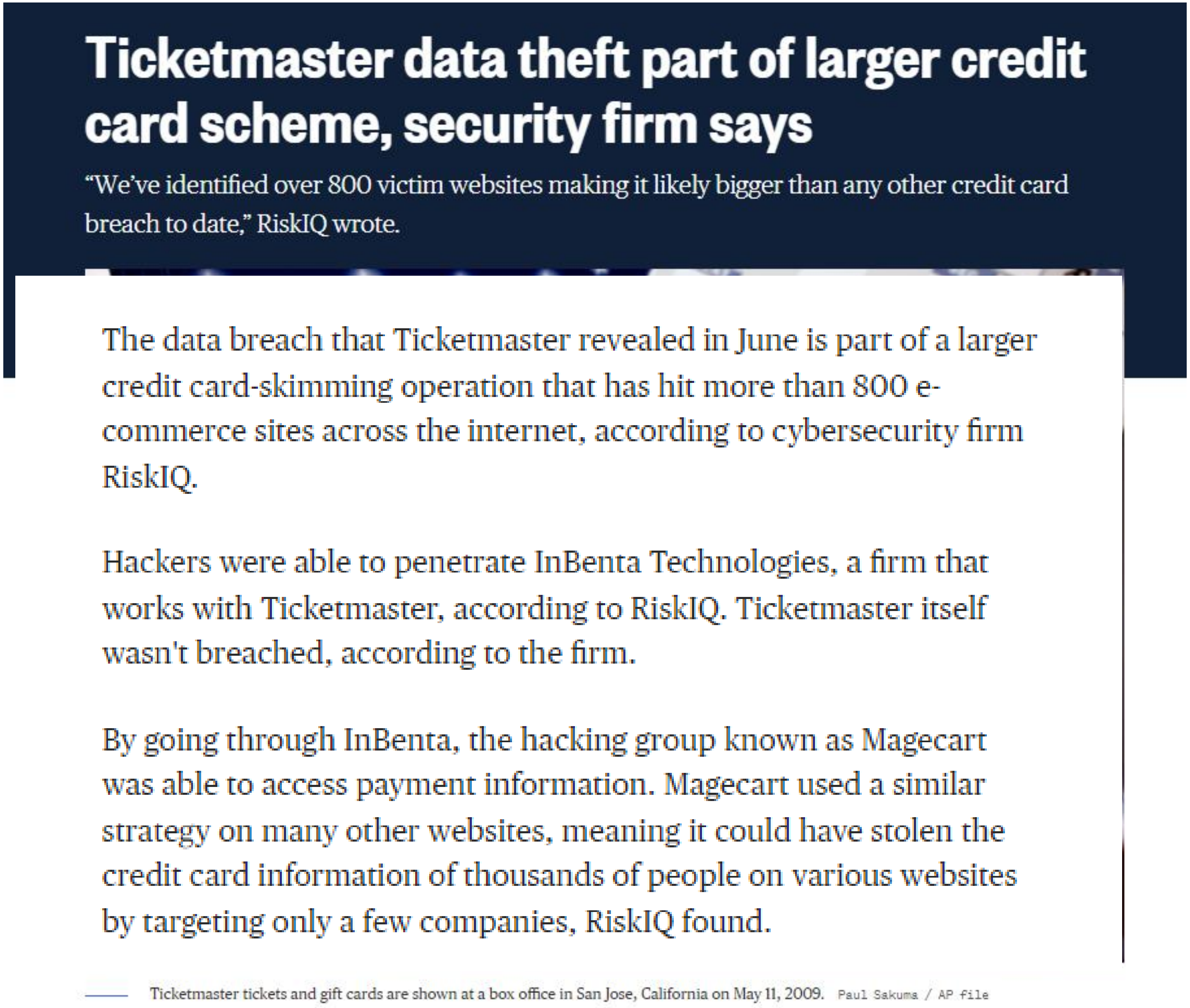


Ticketmaster data theft part of larger credit card scheme, security firm says

“We’ve identified over 800 victim websites making it likely bigger than any other credit card breach to date,” RiskIQ wrote.

ticketmaster
get tickets at ticketmaster.com
NO EXCHANGE EXCEPT AS PROVIDED HEREIN. NO REFUNDS.
EVENT CODE
music sports theater arts&theater family
card

Ticketmaster tickets and gift cards are shown at a box office in San Jose, California on May 11, 2009. Paul Sakuma / AP file



Ticketmaster data theft part of larger credit card scheme, security firm says

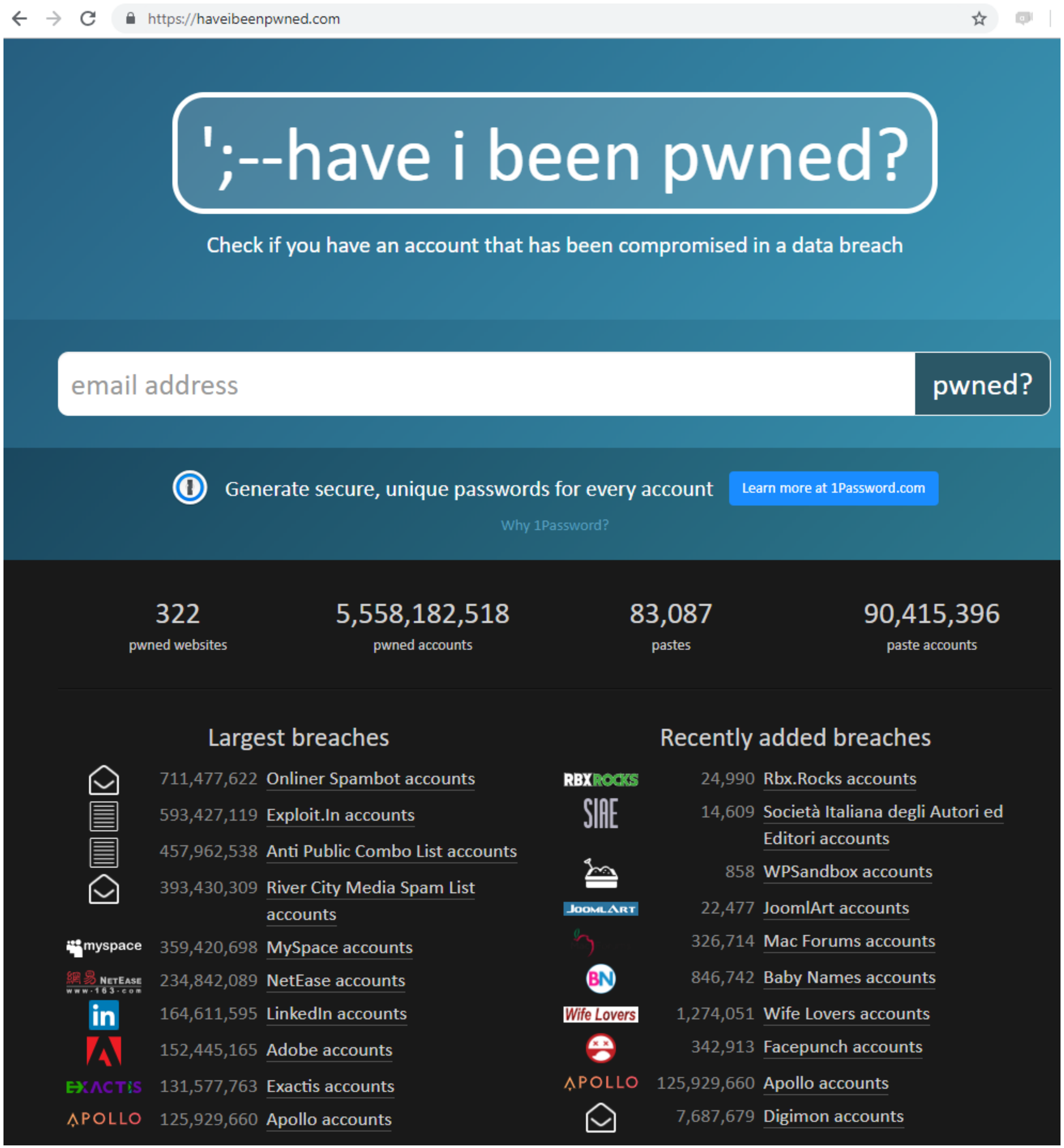
"We've identified over 800 victim websites making it likely bigger than any other credit card breach to date," RiskIQ wrote.

The data breach that Ticketmaster revealed in June is part of a larger credit card-skimming operation that has hit more than 800 e-commerce sites across the internet, according to cybersecurity firm RiskIQ.

Hackers were able to penetrate InBenta Technologies, a firm that works with Ticketmaster, according to RiskIQ. Ticketmaster itself wasn't breached, according to the firm.

By going through InBenta, the hacking group known as Magecart was able to access payment information. Magecart used a similar strategy on many other websites, meaning it could have stolen the credit card information of thousands of people on various websites by targeting only a few companies, RiskIQ found.

Ticketmaster tickets and gift cards are shown at a box office in San Jose, California on May 11, 2009. Paul Sakuma / AP file



https://haveibeenpwned.com

';--have i been pwned?

Check if you have an account that has been compromised in a data breach

Generate secure, unique passwords for every account

322	5,558,182,518	83,087	90,415,396
pwned websites	pwned accounts	pastes	paste accounts

Largest breaches

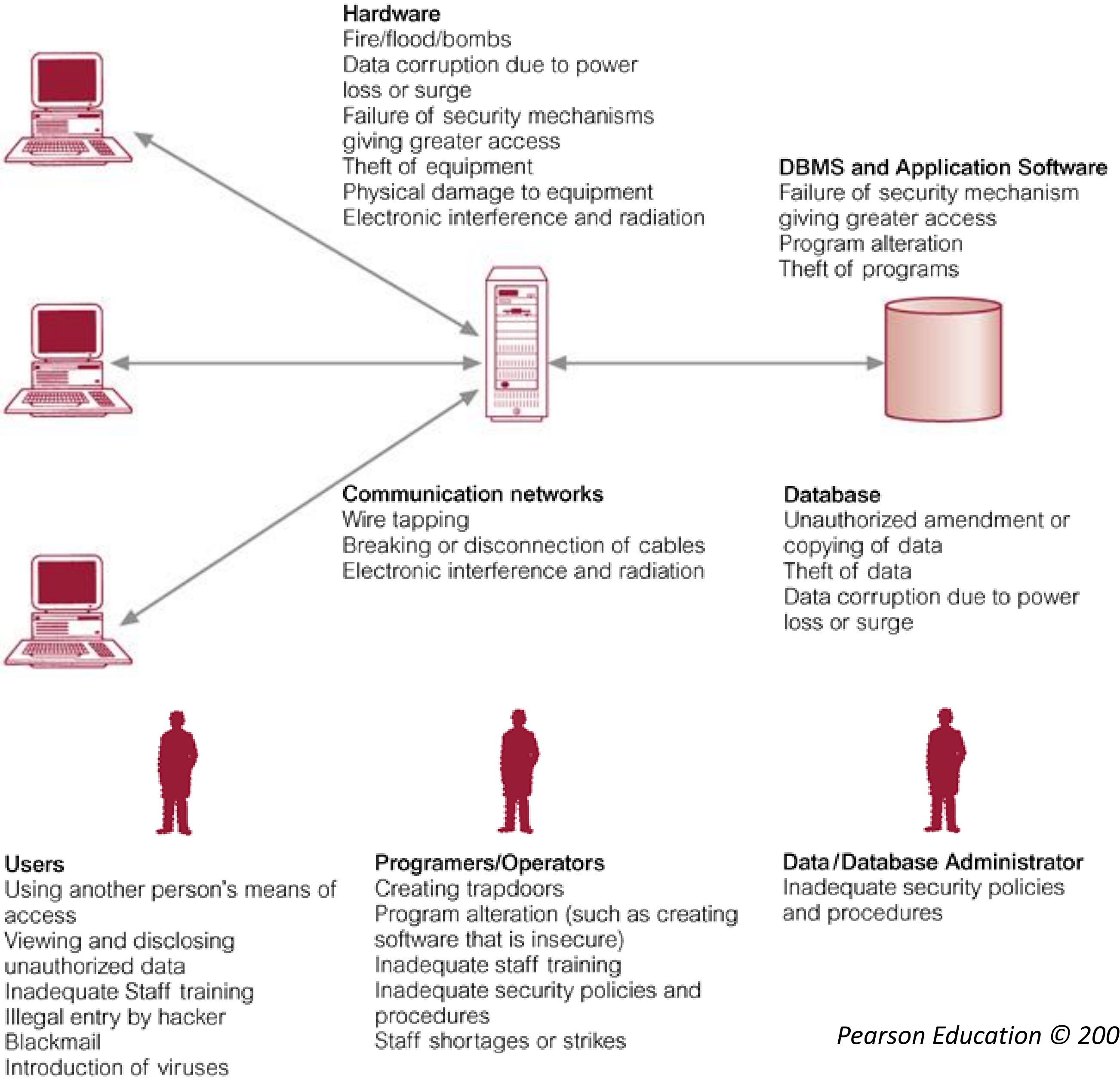
711,477,622	Onliner Spambot accounts
593,427,119	Exploit.In accounts
457,962,538	Anti Public Combo List accounts
393,430,309	River City Media Spam List accounts
359,420,698	MySpace accounts
234,842,089	NetEase accounts
164,611,595	LinkedIn accounts
152,445,165	Adobe accounts
131,577,763	Exactis accounts
125,929,660	Apollo accounts

Recently added breaches

24,990	Rbx.Rocks accounts
14,609	Società Italiana degli Autori ed Editori accounts
858	WPSandbox accounts
22,477	JoomlArt accounts
326,714	Mac Forums accounts
846,742	Baby Names accounts
1,274,051	Wife Lovers accounts
342,913	Facepunch accounts
125,929,660	Apollo accounts
7,687,679	Digimon accounts

Potential Threats

- Theft and fraud
- Loss of confidentiality (secrecy)
- Loss of privacy
- Loss of integrity
- Loss of availability



Hardware
Fire/flood/bombs
Data corruption due to power loss or surge
Failure of security mechanisms giving greater access
Theft of equipment
Physical damage to equipment
Electronic interference and radiation

DBMS and Application Software
Failure of security mechanism giving greater access
Program alteration
Theft of programs

Communication networks
Wire tapping
Breaking or disconnection of cables
Electronic interference and radiation

Database
Unauthorized amendment or copying of data
Theft of data
Data corruption due to power loss or surge

Users
Using another person's means of access
Viewing and disclosing unauthorized data
Inadequate Staff training
Illegal entry by hacker
Blackmail
Introduction of viruses

Programers/Operators
Creating trapdoors
Program alteration (such as creating software that is insecure)
Inadequate staff training
Inadequate security policies and procedures
Staff shortages or strikes

Data/Database Administrator
Inadequate security policies and procedures

Pearson Education © 2009

Computer-based Countermeasures

- Access control
  - Authentication and authorisation
- Views
- Integrity control
- Secure software design
- Encryption
- Hierarchical/distributed storage systems
- Backup and recovery

Outline

- Database security:
  - Threats and countermeasures
- Security in database applications:
  - Robust programming
  - Secure HTTP
  - Sensitive data
  - Passwords
  - SQL injection
- Privileges and roles
- Security in MariaDB

How to Protect System and Users

- Add layers of protection:
  - Make it hard to break into the system:
    - Good passwords – long, …
    - Slow password checking
    - SSL to protect communication within the system
    - Clean response to user upon failures
    - Logging of failures and abnormal activities
    - Robust coding
  - Limit possibilities for damage when having access:
    - Limited possibility for altering data and database – principle of least privilege
    - Password data that is hard to crack
    - Encrypted sensitive data
    - Back up frequently

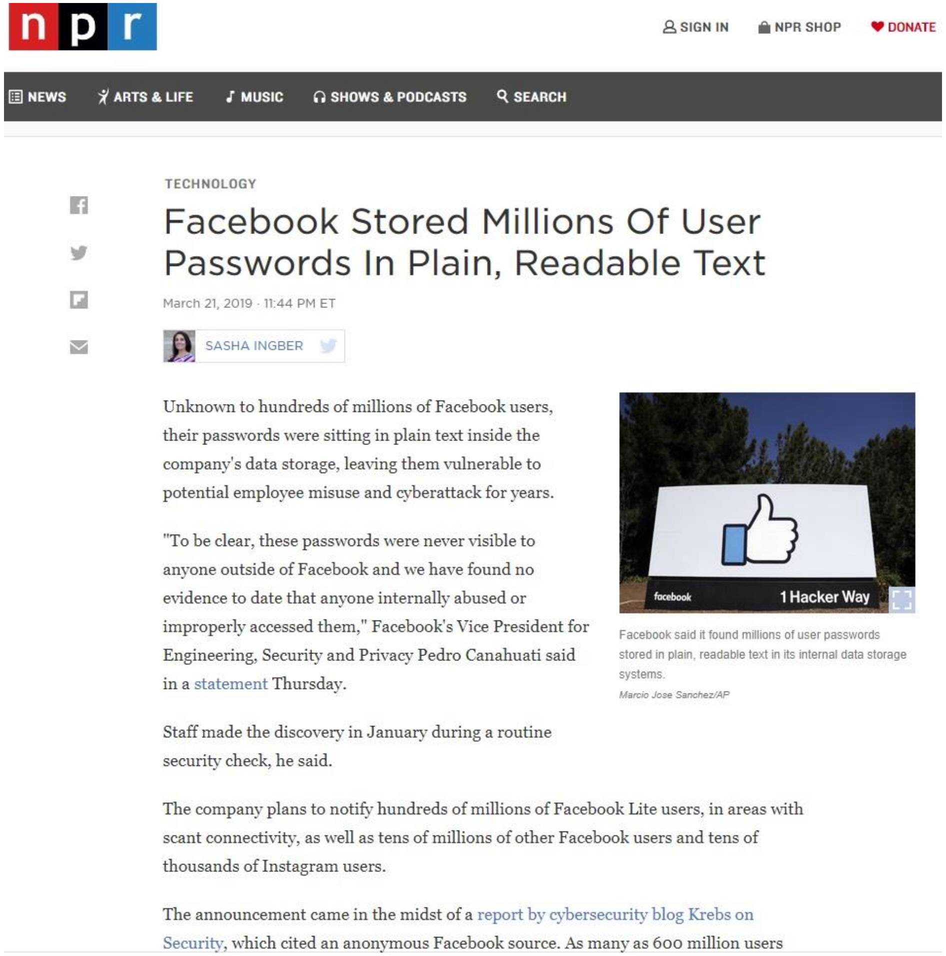
Secure HTTP

- SSL (Secure Socket Layer) can be utilised to encrypt communication between client and server:
  - URL:
    - `HTTPS://...`

Sensitive data

- Database views can be used to limit access to columns containing sensitive data
- Never store sensitive data as plain text in the database
- Encryption – decryption:
  - Text can be stored in encrypted form and decrypted after being retrieved
  - Python ***hashlib*** module

Storing Passwords in a Database (1)



TECHNOLOGY

Facebook Stored Millions Of User Passwords In Plain, Readable Text

March 21, 2019 · 11:44 PM ET

SASHA INGBER

Unknown to hundreds of millions of Facebook users, their passwords were sitting in plain text inside the company's data storage, leaving them vulnerable to potential employee misuse and cyberattack for years.

"To be clear, these passwords were never visible to anyone outside of Facebook and we have found no evidence to date that anyone internally abused or improperly accessed them," Facebook's Vice President for Engineering, Security and Privacy Pedro Canahuati said in a statement Thursday.

Facebook said it found millions of user passwords stored in plain, readable text in its internal data storage systems.

Marcio Jose Sanchez/AP

Staff made the discovery in January during a routine security check, he said.

The company plans to notify hundreds of millions of Facebook Lite users, in areas with scant connectivity, as well as tens of millions of other Facebook users and tens of thousands of Instagram users.

The announcement came in the midst of a report by cybersecurity blog Krebs on Security, which cited an anonymous Facebook source. As many as 600 million users

Storing Passwords in a Database (2)

- Never store passwords as plain text
- Hashing – a one-way encryption:
  - Impossible to decode
  - Useful for checking the correctness of a given value:
    - Correct password
  - MD5, SHA1, SHA256, ...
  - But a lookup table/rainbow table may contain hashes for millions of strings
    - Fast hashing algorithms are a security risk

Storing Passwords in a Database (3)

- Use a strong and slow hashing function
- Add a "salt" to the user-provided password:
  - The salt should be long and random to make it infeasible to generate rainbow tables
  - The hash is computed after adding the salt to the password
  - Then the username, hashed password and the salt can be stored in the database
  - User database

Username	Hashed Password	Random Salt
CrashTV	6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b	1d2a
CrypticHatter	4b227777d4dd1fc61c6f884f48641d02b4d121d3fd328cb08b5531fcacdabf8a	O?1w
UltimateBeast	cd0aa9856147b6c5b4ff2b7dfee5da20aa38253099ef1b4a64aced233c9afe29	LTS)

Storing Passwords in a Database (4)

- Is it safe to store the salt in the database?
- The salt can be added to any place of the password before hashing
- eg- Password required for system at least 10 characters

User submitted password: y0Gp!#A-Hs

Random salt: *_xz

Password+salt option 1: y0*_xzGp!#A-Hs

Password+salt option 2: y0G*_xzp!#A-Hs

Password+salt option 3: y0Gp*_xz!#A-Hs

Password+salt option 4: y0Gp!#A-*_xzHs

Outline

- Database security:
  - Threats and countermeasures
- Security in database applications:
  - Robust programming
  - Secure HTTP
  - Sensitive data
  - Passwords
  - SQL injection
- Privileges and roles
- Security in MariaDB

SQL Injection

- How can it happen?
- Examples of what may happen
- How to avoid

SQL Injection - How can it Happen?

- Exploiting SQL meta-characters:
  - ' (end of string)
  - \ (escape of special character)
  - ; (statement delimiter)
  - -- (start of comment)
- Client input passed directly to the database
- SQL query structure

SQL Injection Example - 1

- Assume a web page listing car models:
  - URL:

```
http://127.0.0.1:5000/GetCarInfo?make=Toyota
```

Passed to the select statement:

```
SELECT make, model FROM car
  WHERE make = Toyota
```

Assume that the user passes:

```
Toyota; SELECT name, password FROM users; --
```

  - What happens then?

```
make, model FROM car
  WHERE make = 'Toyota';
SELECT name, password
  FROM users; --
```

SQL Injection Example - 2

- Assume that the user passes:

```
Toyota'; UPDATE USER SET password='new' WHERE name='CrashTV';
--
```

  - What happens then?

```
SELECT make, model FROM car
  WHERE make = 'Toyota';
UPDATE user
  SET password='new'
  WHERE username='CrashTV'; -- '
```

  - … or if the password is encrypted but may be changed through a link:

```
Toyota'; UPDATE user SET email='hacker@home.com' WHERE
        username='CrashTV'; --
```

SQL Injection Example - 3

- Assume that the user passes:

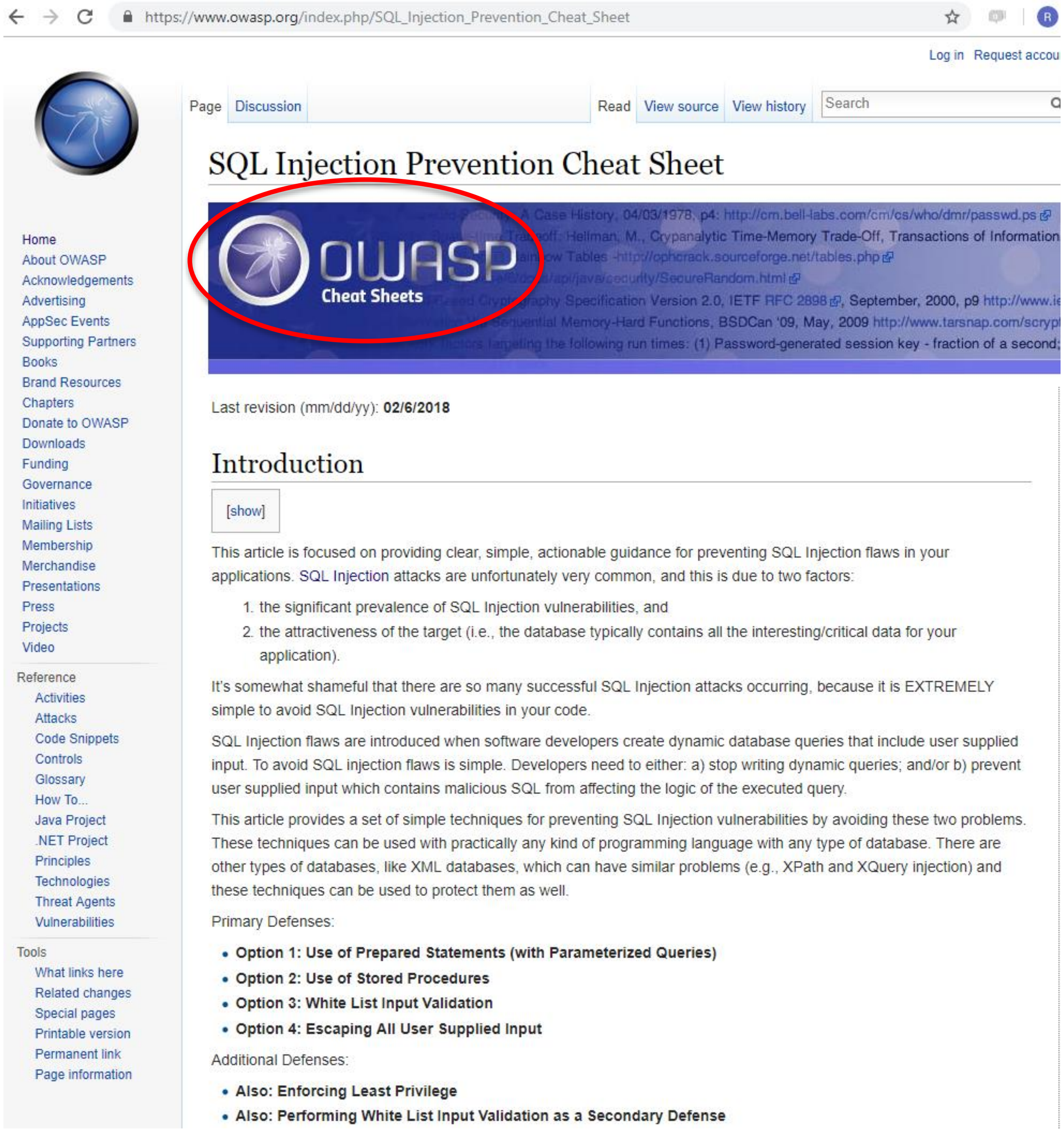
```
Toyota'; DROP TABLE user; --
```

  - What happens then?

```
SELECT make, model FROM car
 WHERE make = ‘Toyota';
DROP TABLE user; -- '
```

SQL Injection – How to Avoid

- Escape/reject any character that is not to be expected:
  - Digits only in numbers
  - Valid email characters only in characters
  - Valid name characters only in names
  - ...
- Use prepared statements!
- Avoid providing the hacker any help and hints



https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Log in Request accou

Page Discussion Read View source View history Search

Home
About OWASP
Acknowledgements
Advertising
AppSec Events
Supporting Partners
Books
Brand Resources
Chapters
Donate to OWASP
Downloads
Funding
Governance
Initiatives
Mailing Lists
Membership
Merchandise
Presentations
Press
Projects
Video

Reference
Activities
Attacks
Code Snippets
Controls
Glossary
How To...
Java Project
.NET Project
Principles
Technologies
Threat Agents
Vulnerabilities

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

SQL Injection Prevention Cheat Sheet

Last revision (mm/dd/yy): **02/6/2018**

Introduction

[show]

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It's somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

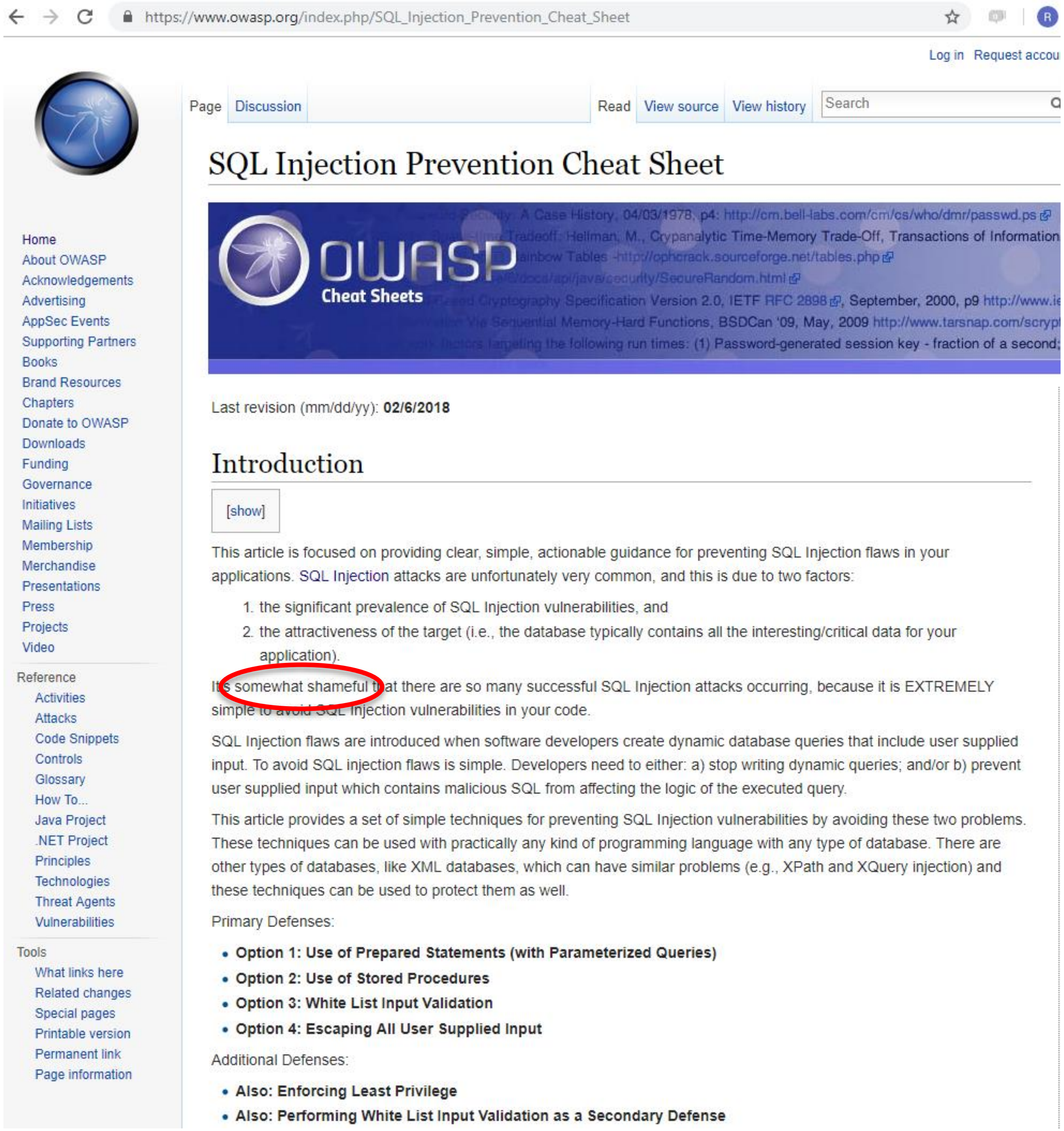
This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Stored Procedures**
- **Option 3: White List Input Validation**
- **Option 4: Escaping All User Supplied Input**

Additional Defenses:

- **Also: Enforcing Least Privilege**
- **Also: Performing White List Input Validation as a Secondary Defense**



SQL Injection Prevention Cheat Sheet

Last revision (mm/dd/yy): **02/6/2018**

Introduction

[show]

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It's somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

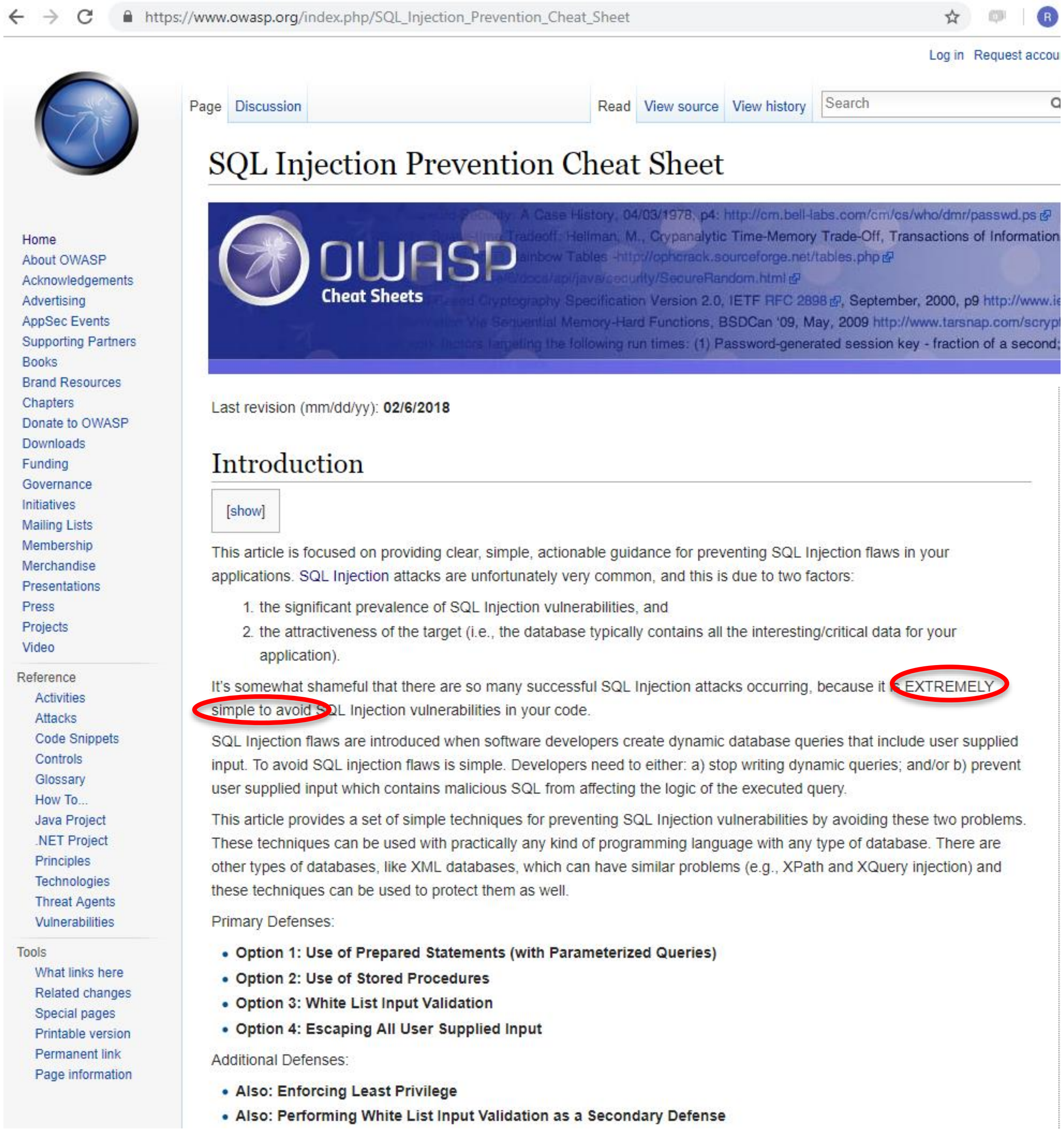
This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Stored Procedures**
- **Option 3: White List Input Validation**
- **Option 4: Escaping All User Supplied Input**

Additional Defenses:

- **Also: Enforcing Least Privilege**
- **Also: Performing White List Input Validation as a Secondary Defense**



Home
About OWASP
Acknowledgements
Advertising
AppSec Events
Supporting Partners
Books
Brand Resources
Chapters
Donate to OWASP
Downloads
Funding
Governance
Initiatives
Mailing Lists
Membership
Merchandise
Presentations
Press
Projects
Video

Reference
Activities
Attacks
Code Snippets
Controls
Glossary
How To...
Java Project
.NET Project
Principles
Technologies
Threat Agents
Vulnerabilities

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

SQL Injection Prevention Cheat Sheet

Last revision (mm/dd/yy): **02/6/2018**

Introduction

[show]

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It's somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

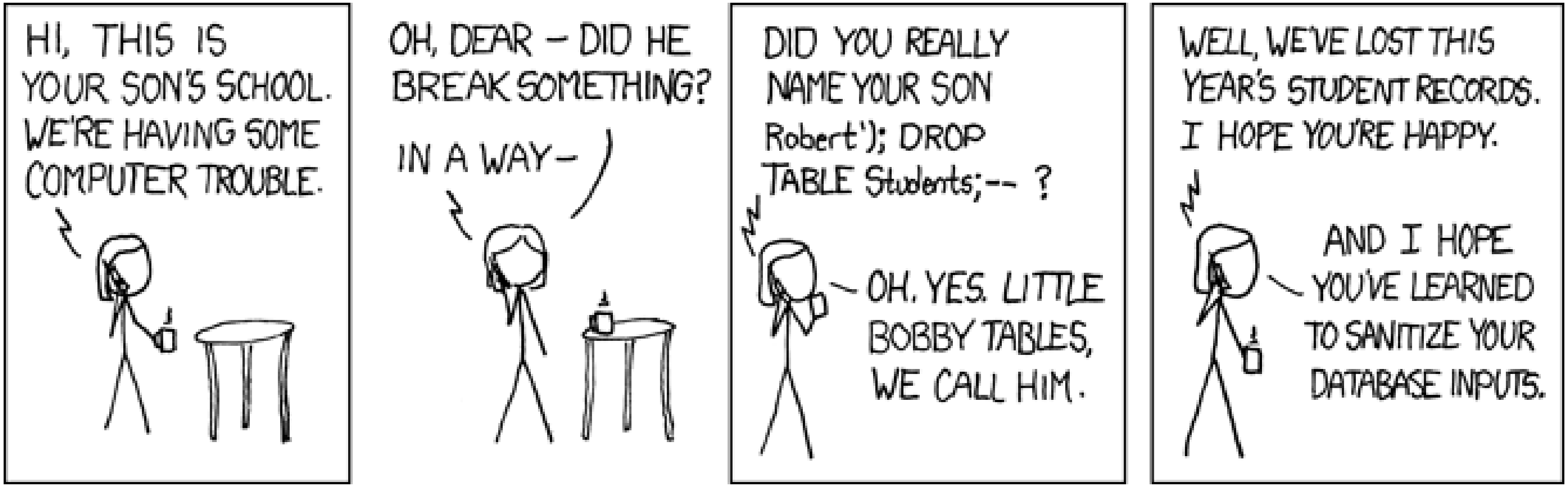
This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.

Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Stored Procedures**
- **Option 3: White List Input Validation**
- **Option 4: Escaping All User Supplied Input**

Additional Defenses:

- **Also: Enforcing Least Privilege**
- **Also: Performing White List Input Validation as a Secondary Defense**

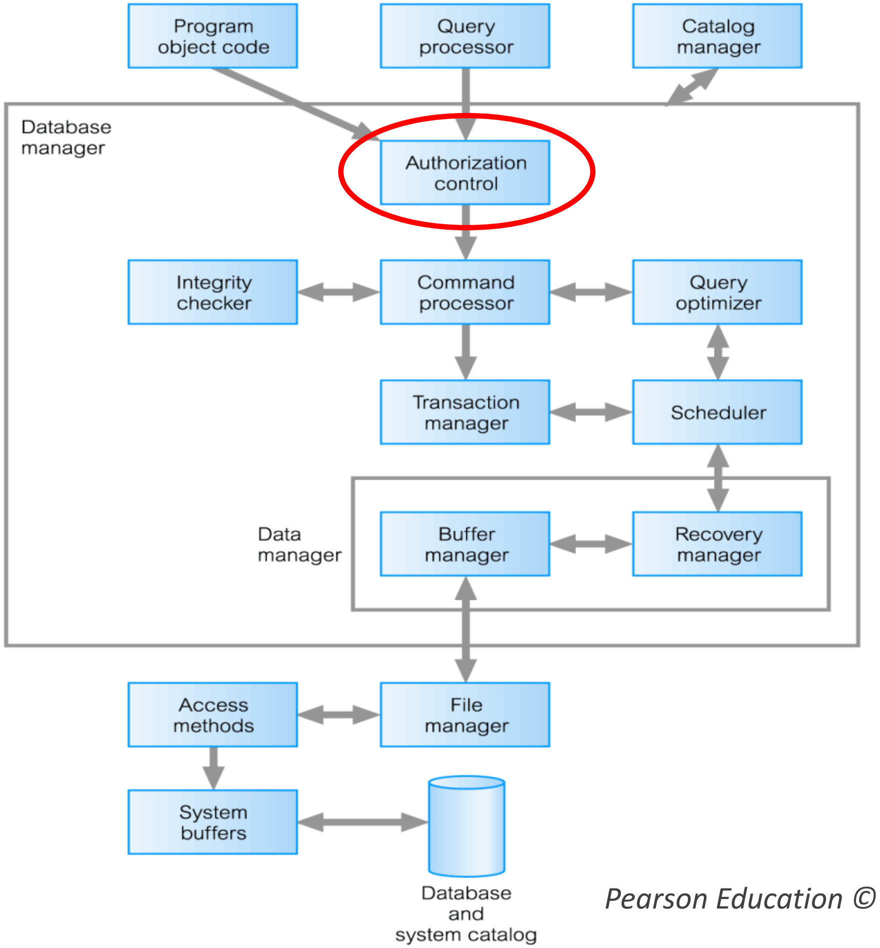

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

https://xkcd.com/327/

Outline

- Database security:
  - Threats and countermeasures
- Security in database web applications:
  - Robust programming
  - Secure HTTP
  - Passwords
  - SQL injection
- Privileges and roles
- Security in MariaDB

Database Manager



Pearson Education © 2009

Authentication vs Authorisation

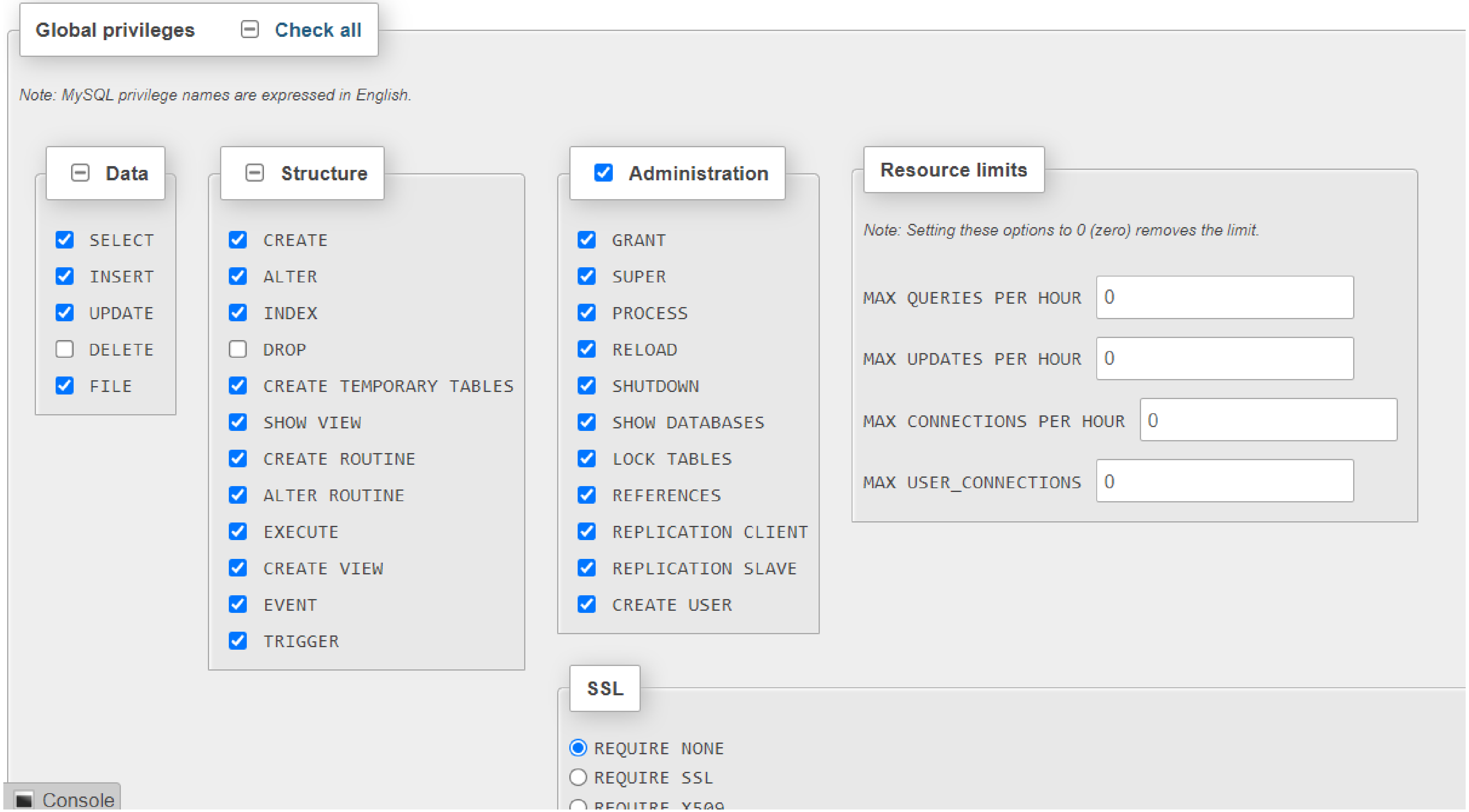
- Authentication:
  - Verifying that the user is who she claims she is
  - Common approach:
    - Username and password
- Authorisation:
  - Verifying that the user is permitted to do what she is trying to do
  - Common approach:
    - Roles and privileges

Database Privileges - 1

- A privilege allows a user to manage:
  - Operations on the database:
    - `CREATE USER`
    - `SHUTDOWN`
    - `...`
  - Operations on individual database objects:
    - `CREATE`
    - `DROP`
    - `ALTER`
    - `DELETE`
    - `INSERT`
    - `SELECT`
    - `UPDATE`
    - `...`

Database Privileges - 2

Edit privileges: User account *'nipuna'@'localhost'*



Global privileges ⊟ Check all

Note: MySQL privilege names are expressed in English.

⊟ Data

- ☑ SELECT
- ☑ INSERT
- ☑ UPDATE
- ☐ DELETE
- ☑ FILE

⊟ Structure

- ☑ CREATE
- ☑ ALTER
- ☑ INDEX
- ☐ DROP
- ☑ CREATE TEMPORARY TABLES
- ☑ SHOW VIEW
- ☑ CREATE ROUTINE
- ☑ ALTER ROUTINE
- ☑ EXECUTE
- ☑ CREATE VIEW
- ☑ EVENT
- ☑ TRIGGER

☑ Administration

- ☑ GRANT
- ☑ SUPER
- ☑ PROCESS
- ☑ RELOAD
- ☑ SHUTDOWN
- ☑ SHOW DATABASES
- ☑ LOCK TABLES
- ☑ REFERENCES
- ☑ REPLICATION CLIENT
- ☑ REPLICATION SLAVE
- ☑ CREATE USER

Resource limits

Note: Setting these options to 0 (zero) removes the limit.

MAX QUERIES PER HOUR	0
MAX UPDATES PER HOUR	0
MAX CONNECTIONS PER HOUR	0
MAX USER_CONNECTIONS	0

SSL

- ● REQUIRE NONE
- ○ REQUIRE SSL
- ○ REQUIRE X509

Database Privileges - 3

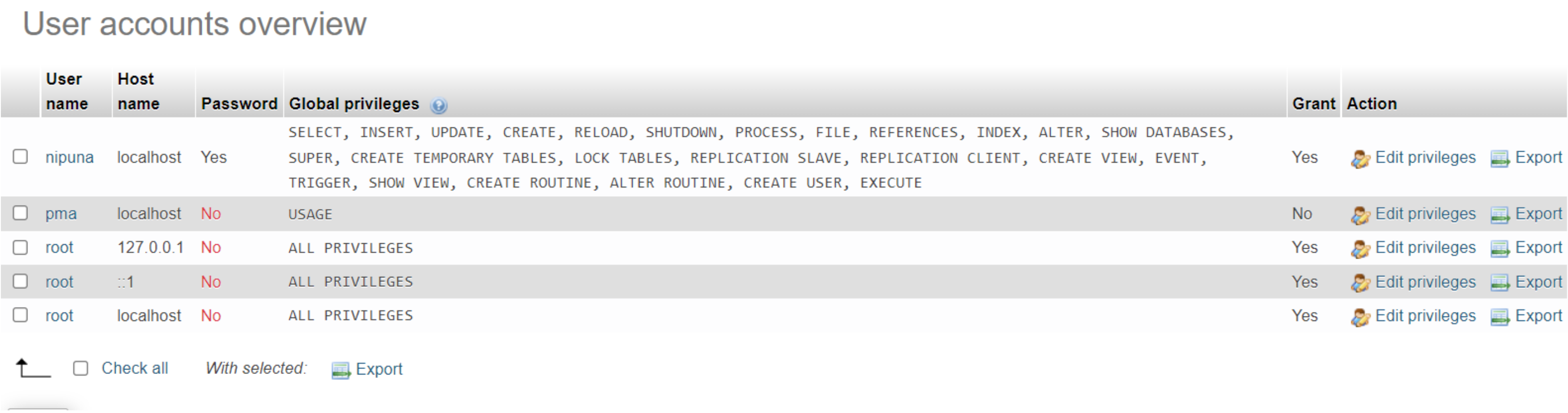
- Privileges can be granted to users:

  -
    ```
    GRANT SELECT, INSERT
          ON sql_exercises.*
          TO 'astudent'@'localhost'
    ```

- And can be revoked from users:

  -
    ```
    REVOKE INSERT
           ON sql_exercises.*
           FROM 'astudent'@'localhost';
    ```

Database Privileges - 4



User accounts overview

	User name	Host name	Password	Global privileges	Grant	Action	
□	nipuna	localhost	Yes	SELECT, INSERT, UPDATE, CREATE, RELOAD, SHUTDOWN, PROCESS, FILE, REFERENCES, INDEX, ALTER, SHOW DATABASES, SUPER, CREATE TEMPORARY TABLES, LOCK TABLES, REPLICATION SLAVE, REPLICATION CLIENT, CREATE VIEW, EVENT, TRIGGER, SHOW VIEW, CREATE ROUTINE, ALTER ROUTINE, CREATE USER, EXECUTE	Yes	Edit privileges	Export
□	pma	localhost	No	USAGE	No	Edit privileges	Export
□	root	127.0.0.1	No	ALL PRIVILEGES	Yes	Edit privileges	Export
□	root	::1	No	ALL PRIVILEGES	Yes	Edit privileges	Export
□	root	localhost	No	ALL PRIVILEGES	Yes	Edit privileges	Export

Database Privileges - 5

- Principle of least privilege:
  - Users should not have wider privileges than necessary, e.g.,
    - Only administrator should be granted admin privileges
    - The database should not run as user root
    - The web app database user should not be the admin user
    - Read-only web apps should not have privileges to update database
    - …

Database Roles

- A role is a named group of privileges
- Individual user can be granted one or more roles:
  - Thereby granted the roles' privileges
- Roles allow for easier and better management of privileges
  - Privileges should preferably be granted to roles

Security in MariaDB

- Encryption and hashing:
  - Encryption functions:
    - `AES_ENCRYPT()`
    - `AES_DECRYPT()`
    - `...`
  - Hashing functions:
    - `MD5()`
    - `SHA1()`
    - `SHA2()`
    - `...`
- Roles are supported

Summary - Security Best Practices

- Employ the principle of least privilege
- Store sensitive data separate from regular data
  - Store sensitive date in encrypted forms
- Never trust what is received from any user
- Use a single authentication failure message regardless of reason
- Use SSL (https) for authentication and sensitive information
  - Use POST when sensitive information is to be submitted
- Hash your passwords with a strong, and slow one-way scheme
  - Salt your hashes, using a separate (random) salt for each string