Security for DB-backed applications

ISYS2120 Data and Information Management

Prof Alan Fekete

University of Sydney

Acknowledge: slides from Uwe Roehm and Alan Fekete, and from the materials associated with reference books (c) McGraw-Hill, Pearson

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Recall: Security Goals

- System and Organisation need to set things up so as to guarantee:
  - **Confidentiality Properties**: Users should not be able to see things which they are not supposed to see.
    - E.g., A student can’t see other students’ grades.
  - **Integrity Properties**: Users should not be able to modify things which they are not supposed to modify
    - This can be broadened to include “users should not be able to perform real-world actions which they are not supposed to be able to do”
    - E.g., Only instructors can assign grades
    - Eg, no-one should be able to get delivery of goods unless they have an account and paid for the goods
  - **Availability Properties**: Users should be able to see and modify things which they are allowed to see and modify.

Some security threats with DB-backed application

- Someone may wish to perform app functionality
  - Eg update a student's grades
- Someone may wish to get information from the system about the domain
  - Eg read about competitor's business plans
- Someone may wish to get information about people (for identity theft)
  - Either exploit it themselves, or sell the information to others
- Someone may wish to use the application as a vehicle for further attacks
  - Eg use the application to install malware on a user's system

The accounts

- Many accounts in various systems
  - Each with its own permissions etc
  - Especial care needed with root/superuser/admin accounts
- There are OS accounts for each machine in the system
  - End-user devices, application server, database server, etc
- There are accounts in the DBMS
- There are accounts known by the application itself
  - Eg customer or user accounts
- Each account will have some way to authenticate and gain access
  - Each account might be attacked!

Whose privileges when code is run?

- Many databases are accessed indirectly
  - End-user does not write and submit SQL, but rather runs a program that (team of) coders have written to perform useful activity
  - Eg a student changes their address through MyUni
- The program can do lots of checking of whether access is appropriate, before sending SQL to dbms
  - Also the program can filter or summarise data, so user does not see everything the program gets from the dbms
- The program may run with its own appropriate level of privilege (or that or the coders), rather than from the end-user who is the source of request
  - Indeed, the end-user may not have a dbms account at all
- Often, the program has quite a lot of privilege, but this is risky if there are mistakes in the code, or if an attacker can obtain the program's credentials [eg if program uses a password which is stored somewhere, and leaked]

Attacks on accounts

- OS accounts: all the usual attacks including installing malware, flaws in the OS, social engineering on legitimate users especially admins
- DB accounts: all the usual attacks, and also a password is often stored in the application code's configuration (so the application can connect); this might be attacked on the machine where the application code runs, or in transmission over the network to the DB server
  - Even if configuration information is encrypted, the encryption key must also be kept on application server (and so can be attacked there)
  - Recall that DBA will be able to see/change anything about DB accounts, so social engineering on the DBA is a concern

Attacks on user accounts in application

- The application will typically store information about its legitimate users in the database (unless SSO etc is used, the app must somehow check the user’s password and so it must store enough to do the check)
  - This can be attacked in many ways: dictionary attack if user has chosen similar password to elsewhere; or another db account who can access the app’s data; or in transit when logging in
    - Importance to use “salt-and-irreversible-hash” when storing password
      - Eg from bcrypt() library
    - Importance to encrypt communication between user and application
  - Also, to maintain session connection, the user typically passes “cookies” with each request to the app; these may be able to be re-used by attacker, to impersonate the user

Attacks through end-user input

- The end-user almost always has places (eg textboxes) where they can provide information to the application, and some of this is likely to be kept in the database
  - Eg user enters their own name, address, contact email
  - Eg user provides search terms
- The application will provide these values as part of database query
  - Dynamic SQL, constructed as application runs, combining a skeleton written by application coder, with user-provided values
- This is an “attack surface” as user may provide values that have malicious purpose
- Famous attack categories:
  - SQL Injection
  - Persistent Cross-Site Scripting
- It is important for application code to sanitize user-provide values to prevent these attacks
  - Application code frameworks typically have methods that do this, provide application coder uses those methods properly!

SQL Code Injection

- **SQL-Injection**
  to infiltrate a SQL database with malicious SQL commands.
  - Can be used to execute SQL statements with elevated privileges or to impersonate another user.
- Injecting SQL via un-checked user input
- Also, attacks with a direct database connection
  - SQL Injection in built-in or user-defined procedures.
  - Buffer overflows in built-in or user-defined procedures.

SQL Injection Attacks

- Web-applications often construct a SQL-statement from separate strings, which may include some provided by the end-user (eg name, address)
  - If a web-application does not thoroughly check the user' s input, in general every database on every operating system is vulnerable.
- Do not assume that the end-user types a simple string; they may maliciously include text that affects the SQL, such as a semicolon and then another command
  - This allows user to provide SQL that will be run with web-app's privileges
- Humour: See https://xkcd.com/327/

WARNING:
Never Query by "String-Concatenation"

Never, never, NEVER ever use Python string concatenation (+) or string parameter interpolation (%) to place variables into a SQL query string!

=> otherwise your program is vulnerable to SQL Injection attacks

```
query ="""SELECT  E.studId  FROM  Enrolled E
          WHERE  E.uosCode = """ + uosCode + \
          " AND E.semester = " + semester
```

string concatenation

```
cursor.execute( query )
```

- *simple approach to construct a variable query*
- *concatenates query from different string-parts*
- *executes the constructed SQL string directly in DBMS*
- ***Warning**: prone to errors and even hacking when input strings allowed to be entered by a user*

Recall: DB-API: Parameterized Queries

- Two (safe) approaches for passing query parameters:
  (because execute() will do any necessary escaping / conversions for parameter markers)

1. Anonymous Parameters

```
    studid = 12345
    cursor.execute(
            "SELECT name FROM Student WHERE sid=%s",
(studid,) )
```

This comma is no mistake, but needed with single parameters

parameter marker

2. Named Parameters

```
    studid = 12345
    cursor.execute(
        "SELECT name FROM Student WHERE sid=%(sid)s",
{'sid': studid} )
```

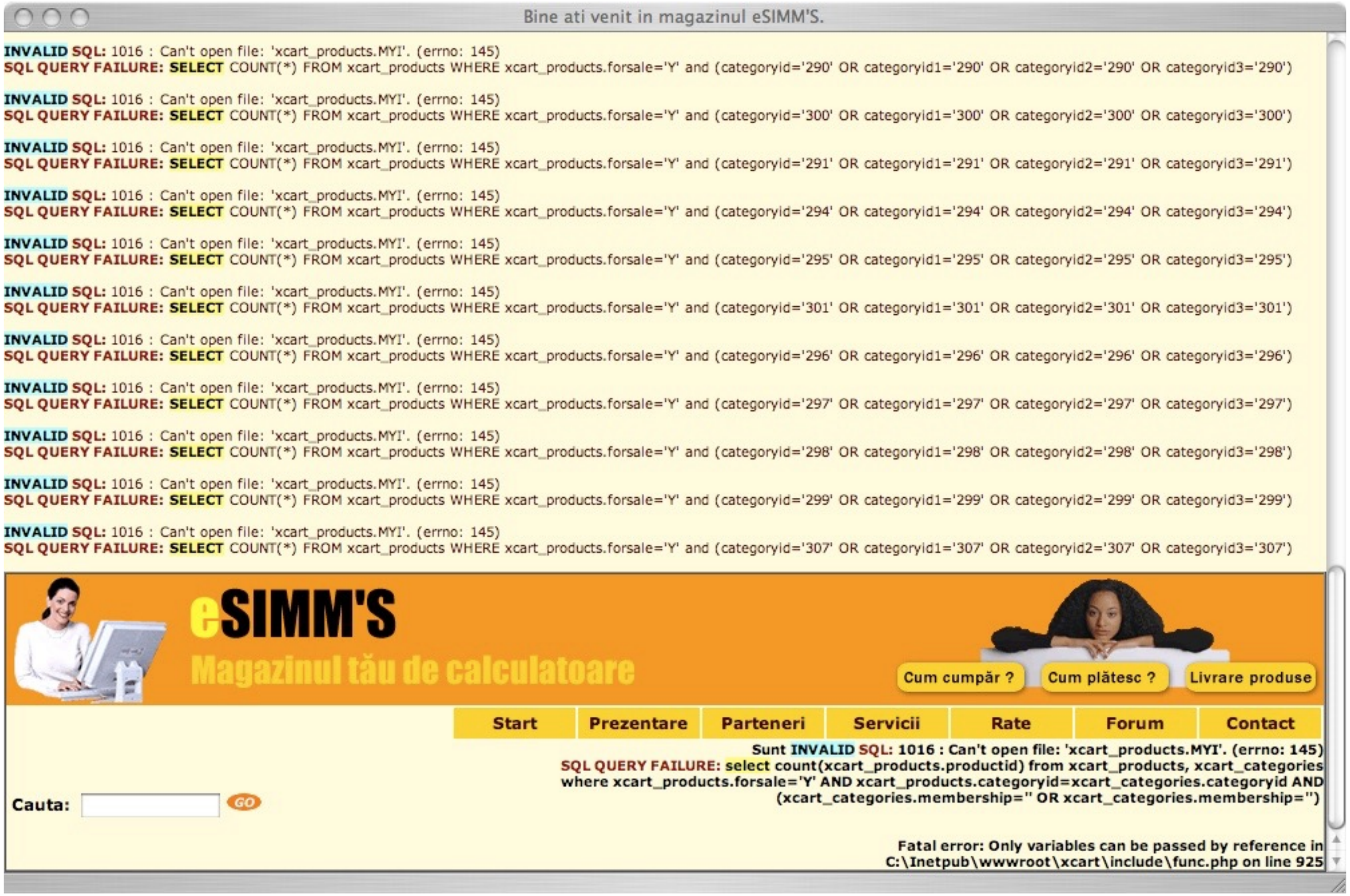
named parameter marker

Persistent XSS

- Persistent cross-site scripting (XSS) is where an attacker puts values into the database, that are sent to another user by the application, and impact the working of the target user's browser
  - Eg to display deceptive material, or introduce malware, etc
- Typically, the value includes characters that the target browser may treat as Javascript commands etc
- Central defence is for application to use templates for the html it sends, which escape or check data from the database before combining it into the page to be rendered

Information leakage through db errors

- If the dbms returns an error from a query, this often includes a lot of information about the situation
  - If the application simply displays the error exception, the end-user could learn a lot they were not supposed to know (and also, be quite confused!)
  - So always craft error messages from application to end-user, to be clear and secure
- Also, be sure to disable “debug” mode in the dbms, as this might allow attacker to interact with the platform



Bine ati venit in magazinul eSIMM'S.

```
INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='290' OR categoryid1='290' OR categoryid2='290' OR categoryid3='290')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='300' OR categoryid1='300' OR categoryid2='300' OR categoryid3='300')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='291' OR categoryid1='291' OR categoryid2='291' OR categoryid3='291')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='294' OR categoryid1='294' OR categoryid2='294' OR categoryid3='294')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='295' OR categoryid1='295' OR categoryid2='295' OR categoryid3='295')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='301' OR categoryid1='301' OR categoryid2='301' OR categoryid3='301')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='296' OR categoryid1='296' OR categoryid2='296' OR categoryid3='296')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='297' OR categoryid1='297' OR categoryid2='297' OR categoryid3='297')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='298' OR categoryid1='298' OR categoryid2='298' OR categoryid3='298')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='299' OR categoryid1='299' OR categoryid2='299' OR categoryid3='299')

INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: SELECT COUNT(*) FROM xcart_products WHERE xcart_products.forsale='Y' and (categoryid='307' OR categoryid1='307' OR categoryid2='307' OR categoryid3='307')
```

```
Sunt INVALID SQL: 1016 : Can't open file: 'xcart_products.MYI'. (errno: 145)
SQL QUERY FAILURE: select count(xcart_products.productid) from xcart_products, xcart_categories
where xcart_products.forsale='Y' AND xcart_products.categoryid=xcart_categories.categoryid AND
(xcart_categories.membership='' OR xcart_categories.membership='')

Fatal error: Only variables can be passed by reference in
C:\Inetpub\wwwroot\xcart\include\func.php on line 925
```

Use high-quality libraries

- Programmers are often tempted to write their own code for security (encryption, checking, etc)
  - This is very dangerous
  - There are many subtleties, it’s easy to get it wrong
- Instead code with well-tested libraries
  - And know how to use the capabilities they have

Protecting a Web Database

- In application coding:
  - Consider how user account information is stored in db, esp don’t store plain-text passwords
  - Be careful to use execute() function with parameter markers, so you sanitize all parameters which can end up in SQL statements
  - Be careful to use templates for HTML generation
  - Never return database error messages to end-users
- In DBMS: Restrict the privileges of the user/role of the web application
- On app server: check where db connection information is kept, and how protected (including db account password)
- Everywhere: Patch, patch, patch ;-)