Input validation

- Lack of input validation is the most commonly exploited vulnerability
- Many variants of attacks that exploit this
  - buffer overflows – "C(++) injection"
    - possibly via format string attacks and integer overflow attacks
  - Command injection
  - SQL injection
  - XSS (Cross site scripting) - "script injection"
  - ...

Input validation

- Buffer overflows
  - format string attacks
  - integer overflow
- Command injection
- SQL injection
- XSS
- File name injection
- General remarks about input validation

Command injection (in a CGI script)

- A CGI script might contain

  cat *thefile* | mail *clientaddress*

- An attack might enter email address

  pippo@di.uniroma1.it | rm -rf

- What happens then ?

  cat thefile | mail pippo@di.uniroma1.it | rm -rf/

- Can you think of countermeasures ?
  - validate input
  - reduce access rights of CGI script (*defense in depth*)
  - maybe we shouldn't use such a scripting languages for this?

Command injection (in a C program)

Code that uses the system interpreter to print to a user-specified printer might include

```
char buf[1024];
snprintf(buf,"system lpr -P %s",printer_name,
          sizeof(buf)-1);
system(buf);
```

This can be attacked in the same way; entering

      miro;xterm&

is less destructive and more interesting than **...;rm -fr /**

Command injection

- Vulnerability: many API calls and language constructs in many languages are affected, eg
  - C/C++ `system(), execvp(), ShellExecute(), ..`
  - Java `Runtime.exec(), ...`
  - Perl `system, exec, open, `, /e, ...`
  - Python `exec, eval, input, execfile, ...`
  - ...
- Countermeasures
  - validate all user input
    - whitelist, not blacklist
  - run with minimal privilige
    - doesn't prevent, but mitigates effects

Input validation

- Buffer overflows
  - format string attacks
  - integer overflow
- Command injection
- SQL injection
- XSS
- File name injection
- General remarks about input validation

SQL injection

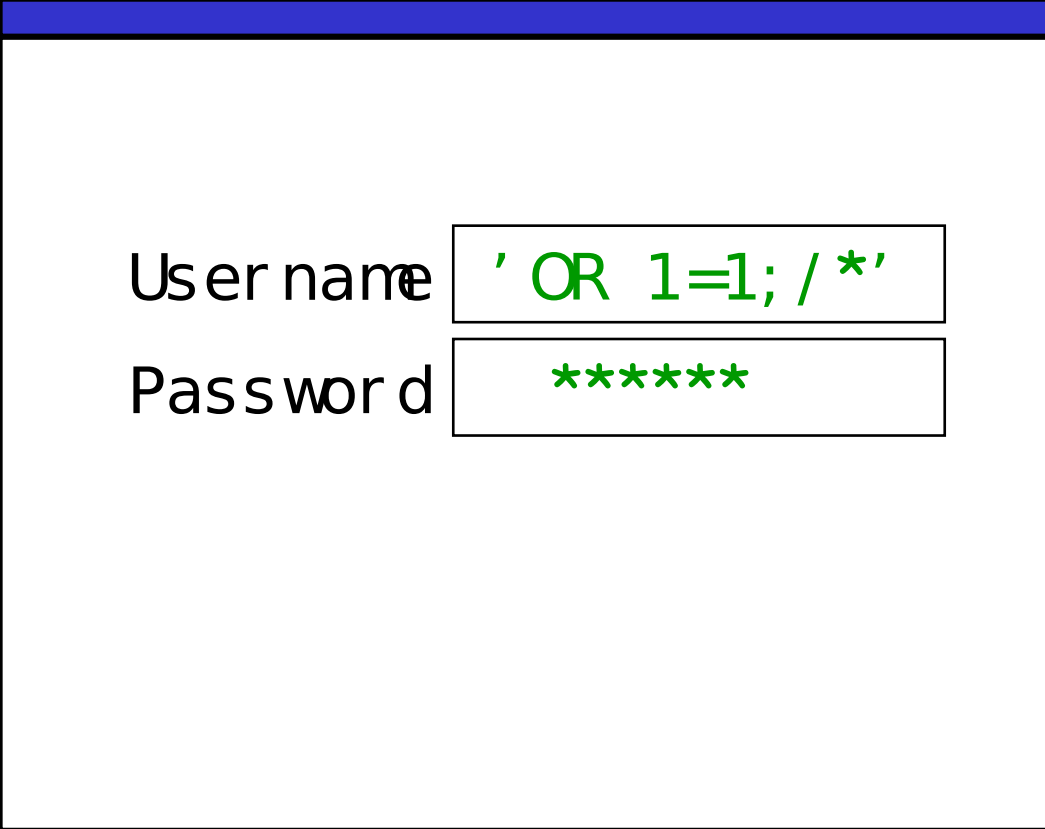Username pippo

Password ******

SQL injection

```
$result = mysql_query(
    "SELECT * FROM Accounts".
    "WHERE Username = '$username'".
    "AND Password = '$password';");
if (mysql_num_rows($result)>0)
        $login = true;
```

SQL injection

Resulting SQL query

```
SELECT * FROM Accounts
WHERE Username = 'pippo'
AND Password = 'secret';
```

SQL injection

Username `'OR 1=1;/*'`
Password `******`

SQL injection

Resulting SQL query

```
SELECT * FROM Accounts
WHERE Username = '' OR 1=1;/*'
AND Password = 'secret';
```

SQL injection

Resulting SQL query

```
SELECT * FROM Accounts
WHERE Username = '' OR 1=1;
/*'AND Password = 'secret';
```

Oops!

SQL injection

- Vulnerability: any application in any programming language that connects to SQL database

  - if it uses dynamic SQL

  - NB typical books such as "PHP & MySQL for Dummies" contain examples with SQL injection vulnerabilities!

- Note the common theme to many injection attacks: contatenating strings, some of them user input, and then interpreting, rendering, or executing the result

  is a VERY BAD IDEA

Avoiding SQL injection: Prepared Statement

Vulnerable:

```
String updateString = "SELECT * FROM Account
WHERE Username" + username + " AND Password = "
+ password;
stmt.executeUpdate(updateString);
```

Not vulnerable:

```
PreparedStatement login =
con.preparedStatement("SELECT * FROM Account
         WHERE Username = ? AND Password = ?" );
login.setString(1, username);
login.setString(2, password);
login.executeUpdate();
```

bind variable

aka parameterised query

Similar: Stored Procedures

Stored procedure in Oracle's PL/SQL

```
CREATE PROCEDURE login
    (name VARCHAR(100), pwd VARCHAR(100)) AS
DECLARE @sql nvarchar(4000)
SELECT @sql =' SELECT * FROM Account WHERE
username=' + @name + 'AND password=' + @pwd
EXEC (@sql)
```
called from Java with

```
CallableStatement proc =
 connection.prepareCall("{call login(?, ?)}");
proc.setString(1, username);
proc.setString(2, password);
```

Some observations

- Other issues – besides security - in discussions about prepared statements, stored procedures, bind variables, ...
  - efficiency
  - bandwidth between web-app and database
  - stored procedures allow common fixed interface to several web-apps

- Moral of the story: check the details for your configuration (language, database system) and your chosen solution!

- Open question: Why is SQL injection still a problem???
  - NB Top vulnerability in OWASP Top 10

  Why doesn't everyone use parameterised queries???

variation: Database Command Injection
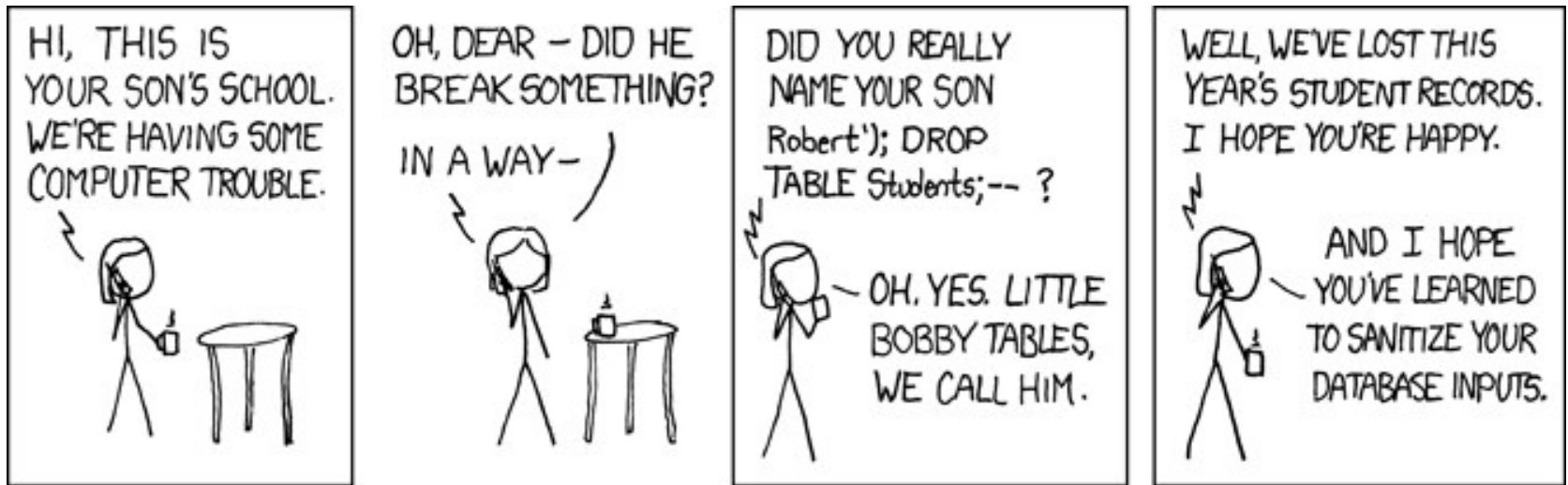
- injecting database command with ;
  not manipulating SQL query with `
- highly dependent on infrastructure, eg
  - each database has its own commands
    - eg. Microsoft SQL Server has `exec master.dbo.xp_cmdshell`
  - some configurations don't allow use of ;
    - eg Oracle database accessed via Java or PL/SQL

21

variation: Function Call Injection

- Oracle SQL has > 1000 built-in functions that can be used inside stored procedures, eg `TRANSLATE`
    - `TRANSLATE('acadaa', 'abcd', 'ABCD') = 'ACADAA'`

- Arguments of such functions may be poisoned with other functions, eg

    ```
    SELECT TRANSLATE('user input'', 'abcd', 'ABCD')
            FROM ...
    ```
    can become
    ```
    SELECT TRANSLATE(''||UTL_HTTP.REQUEST(http://..
                                    ...) ||'',
                  'abcd', 'ABCD')
    FROM ...
    ```

    `UTL_HTTP` does HTTP request directly from Oracle database, which is probably running *behind* the firewall...

22

Countermeasures to SQL injection

- input validation
- use prepared statements aka parameterised queries with bind variables
  - not string concatentation
  - or stored procedures, *if* these are safe
- use language/system level countermeasures
  - eg magic quotes in PHP
- apply principle of least privilige
  - ie. minimise rights of web application
- know what you're doing: find out the threats & countermeasures for your specific configuration, programming language, database system…

Finding such SQL injection vulnerabilites?

- Google code search!

- Eg

   lang:php "WHERE username='$_"

   ie. http://google.com/codesearch?
   hl=en&start=10&sa=N&filter=0&q=lang:php+
   %22WHERE+username%3D%27%24_%22

Input validation

- Buffer overflows
  - format string attacks
  - integer overflow
- Command injection
- SQL injection
- XSS (Cross site scripting)
- File name injection aka path traversal
- General remarks about input validation

File name injection

- user-supplied file name may be
  - existing file   `../../../etc/passwd`
  - not really a file   `/var/spool/lpr`
  - file the user can access in other ways

    `/mnt/usbkey, /tmp/file`
- this may break
  - *confidentiality* (leaking information to the user)
  - *integrity* (eg. of file or system)
  - *availability* (eg. trying to open print device for reading)

File name injection

- File names constructed from user input – eg by string concatenation – are suspect too

  - Eg what is

    ```
    "/usr/local/client-info/" ++ name
    ```
    if `name` is `../../../etc/passwd` ?

- aka directory traversal attack
- validating file names is difficult: reuse existing code and/or use chroot jail

Input validation

- Buffer overflows
  - format string attacks
  - integer overflow
- Command injection
- SQL injection
- XSS (Cross site scripting)
- File name injection
- General remarks about input validation

Input validation in general

- Input validations problems are *the* most common vulnerabilities

- Never *ever* trust any user input
  - Apart from generic risks dicussed so far (command, SQL, XSS, filenames,...), there will be additional input risks specific to an application

- Beware of *implicit assumptions* on user input
  - eg, that usernames only contain alphanumeric characters

- Think like an attacker!
  - think how you might abuse a system with weird input

Input validation problems: _prevention_

find out about potential vulnerabilities:

use community resources to find out vulnerabilities of the system/language used

avoid use of unsafe constructs, if possible

make sure all input is validated

at clear _choke-point_s in code

when doing input validation

- use white-lists, not black-lists
  - unless you are 100% sure your black-list is complete
- reuse existing input validation code known to be correct

Input validation problems: *detection*

- testing

  test with inputs likely to cause problems
  - for buffer overflow, long inputs (fuzzing)
  - for SQL injection, inputs with fragments of SQL commands
  - ...

  There are some tools that can help, eg webscarab for XSS

  Note: web-application returning a page with SQL error message is a bad sign...

Input validation problems: *detection*

- testing

  test with inputs likely to cause problems
  - for buffer overflow, long inputs (fuzzing)
  - for SQL injection, inputs with fragments of SQL commands
  - ...

  There are some tools that can help, eg webscarab for XSS
  Note: web-application returning a page with SQL error message is a bad sign...

- tainting
  - effectively typing, with runtime checking or static analysis (more precisely, data flow analysis)
    - eg SA_PRE(Tainted=SA_True) in PREfast

- code reviews, possibly using static analysis

Input validation – Conclusion

- Lack of input validation no #1 problem
  - in various guises

- Never trust user input!

- Think about, test, and detect malicious inputs!

- Find out about the vulnerabilities of specific language, platform,.. and about countermeasures

OWASP Top 10 – 2010 release

- injection flaws
- cross site scripting
- broken access control and session management
- insecure direct object reference
- cross site request forgery
- security misconfiguration (new)
- failure to restrict URL access
- unvalidated redirects & forwards (new)
- insecure cryptographic storage
- insecure communications

See OWAPS.org.

2004 edition still mentioned buffer overflows, but 2007 edition no longer did

19 Deadly sins of software security
[Howard, LeBlanc, Viega, 2005]

- buffer overruns
- format string problems
- integer overflows
- SQL injection
- command injection
- failing to handle errors
- XSS
- failing to protect network traffic
- use of magic URLs or hidden form fields
- improper use of TLS, SSL

- weak passwords
- failing to store & protect data securely
- information leakage
- improper file access
- trusting network name resolution
- race conditions
- unauthenticated key exchange
- weak random numbers
- poor usability

blue ones are input problems

49

24 Deadly sins of software security
[Howard, LeBlanc, Viega, *2009*]

New deadly sins in 2009 edition of the book

- C++ catastrophes

- not updating easily

- executing code with too much privilige

- sins of mobile code

Also, sins collected in 4 categories (web-apps, implementation, cryptographic,networking)

These Top n lists are nice, BUT..

- There is more to security than knowing the latest top 10 of common security vulnerabilities
  - esp. thinking about & minimizing potential problems in the design phase

- Some efforts at classifications of the security vulnerabilities
  - but if you've seen enough of them, you quickly spot some common features

Classification of Software Security Errors

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Errors
6. Code Quality
7. Encapsulation
*. Environment

[Katrina Tsipenyuk, Brian Chess, Gary McGraw, Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors]

One of the CWE classifications (cve.mitre.org)