CS390S, Week 7: Input Validation and SQL Injection

Pascal Meunier, Ph.D., M.Sc., CISSP
October 4, 2006
Developed thanks to the support of Symantec Corporation, NSF SFS Capacity Building Program (Award Number 0113725) and the Purdue e-Enterprise Center

Copyright (2004) Purdue Research Foundation. All rights reserved.
Input Validation

- Why validate?
  - The purposes of input validation

- Validate what, where?
  - Boundaries
    - Trust boundaries
    - Data model boundaries
    - Subsystem or module boundaries
  - Data models
    - Encodings
    - Tab-separated
    - XML
Goals of Input Validation (white list approach)

- Enforce program correctness
- Preserve an application's invariants
  - Item prices are always 0 or greater
  - Money is never created or lost (double entry accounting)
  - If you don't know what the invariants are, you can't perform complete input validation
- Enforce or verify design assumptions
  - Assumptions need to be known and explicitly stated
  - Formula used to calculate breaking distance
    - Only holds with speeds smaller than X
    - and altitude must be less than Y
Preventative Input Validation (black list approach)

- Prevent unexpected behavior
  - How do you prevent what you didn't expect?
- Prevent vulnerabilities and exploits (policy violations)
  - e.g., code injection
  - Can you enumerate all possible issues and prove that you prevent them?
    - Without forgetting any?
Goal: trick program into executing an attacker’s code by clever input construction that mixes code and data

Mixed code and data channels have special characters that trigger a context change between data and code interpretation

- The attacker wants to inject these meta-characters through some clever encoding or manipulation, so supplied data is interpreted as code
Basic Example by Command Separation

- `cat > example`
  - `#!/bin/sh`
  - `A=$1`
  - `eval "ls $A"`

- Permissions for file "confidential" before exploit:
  - `% ls -l confidential`
    - `-rwxr-x---  1 pmeunier  pmeunier confidential`

- Allow execution of "example":
  - `% chmod a+rx example`

- Exploit (what happens?)
  - `% ./example ".;chmod o+r *"`
Results

- Inside the program, the eval statement becomes equivalent to:
  ```
eval "ls .;chmod o+r *
  ```
- Permissions for file "confidential" after exploit:
  ```
  % ls -l confidential
  -rwrxr-xr-- 1 pmeunier pmeunier confidential
  ```
- Any statement after the ";" would also get executed, because ";" is a command separator.
- The data argument for "ls" has become code!
A Vulnerable Program

- int main(int argc, char *argv[], char **envp)
  {
    char buf [100];
    buf[0] = '\0';
    snprintf(buf, sizeof(buf), "grep %s text", argv[1]);
    system(buf);
    exit(0);
  }

What happens when this is run?
% ./a.out `./script`

http://www.cerias.purdue.edu
Answer

- The program calls
  - system ("grep `./script` text");
  - You can verify by adding "printf( "%s", buf)" to the program

- So we could make a.out execute any program we want
  - Imagine that we provide the argument remotely
  - What if a.out runs with root privileges?
Mixed Data and Code Examples

- Wrappers to system calls
  - Command vs arguments
  - subshells, command substitution ("\`")
  - other shell metacharacters
- HTML vs JavaScript
  - "<script>"
  - "on eventname"
- Format Strings
  - Special format specifiers
- SQL (Simple Query Language for databases)
The Input Cleansing Idea

- Model the expected input
  - Discard what doesn't fit (e.g., metacharacters)

- Intuitive Approach
  - Block or escape all metacharacters
    - but what are they?
  - Problems:
    - Character encodings
      - octal, hexadecimal, UTF-8, UTF-16...
    - Obfuscation
    - Escaped characters that can get interpreted later
    - Engineered strings such that by blocking a character, something else is generated
Input Cleansing and Sanitization

- Error prone
- Complex
- May be insufficient (validation still needs to be performed) or too crude (loss of functionality)
- Black List approach
- Instead of trying to pick valid parts of the input and to recover from attacks in the input, it is safer to simply reject input identified as incorrect (and potentially malicious)
Defending Against Code Injection

- **Architecture**: separate code from data
  - Transmit, receive and manipulate data using different channels than for code

- **Aim for program correctness (White List)**
  - Identify boundaries
  - Identify data type, range and organization
  - Identify calling models (e.g., format strings, and who is responsible for what)
  - Identify assumptions and invariants
  - Identify data dependencies
  - Verify and translate data models, enforce assumptions and invariants at boundaries, and check data dependencies
SQL Injection

- SQL uses single and double quotes to switch between data and code.
- Semi-colons separate SQL statements
- Example query:
  - "UPDATE users
    SET prefcolor='red'
    WHERE uid='joe';"
- This command could be sent from a web front-end to a database engine.
- The database engine then interprets the command
Web applications typically dynamically generate the necessary database commands by manipulating strings.

Example query generation:

```sql
$q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
```

Where the value of "$INPUT[color]" would be originating from the client web browser, through the web server.

And where the value for "$auth_user" would have been stored on the server and verified through some authentication scheme.
Forms in client browsers return values to the web server through either the POST or GET methods

- "GET" results in a url with a "?" before the values of the form variables are specified:
  - http://www.example.com/script?color=red
  - The value of "$INPUT[color]\" is set to "red" in the script

"GET" urls are convenient to hack, but there isn't any significant difference in the security of either "GET" or "POST" methods because the data comes from the client web browser regardless and is under the control of the remote attacker
The SQL Table

- Tables are used to store information in fields (columns) in relation to a key (e.g., "uid")
- What other fields could be of interest?
- CREATE TABLE users (prefcolor varchar(20),
  uid VARCHAR(20) NOT NULL,
  privilege ENUM('normal', 'administrator'),
  PRIMARY KEY (uid)) ;
What if we could make the web server generate a query like:

```
"UPDATE users
SET prefcolor='red',
privilege='administrator'
WHERE uid='joe';"
```

Can we engineer the value of "color" given to the web server so it generates this query?

- Note how code and data are mixed in the same channel
  - Better database interfaces provide separate channels
    - Java prepared statements
    - Stored procedures
Malicious HTTP Request

- http://www.example.com/script?color=red', privilege='administrator

- The "color" input is then substituted to generate SQL:
  - $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'';

- It gives the query we wanted!
Results

- Joe now has administrator privileges.
Adding Another SQL Query

- Let's say Joe wants to run a completely different query:
  - "DELETE FROM users"
    - This will delete all entries in the table!
- How can the value of "color" be engineered?
Malicious HTTP Request

- http://www.example.com/script?color=red%3Bdelete+from+users%3B
  - %3B is the url encoding for ";"

- What happens when the "color" input is used to generate SQL?
  - $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
UPDATE users
SET prefcolor='red';
delete from users;
WHERE uid='$auth_user'";

- The last line generates an error, but it's already too late; all entries have been deleted.
- The middle query could have been anything
FAQs

- Couldn't the database have a separate account for "Joe" with only the privileges he needs (e.g., no delete privilege)?
  - In theory yes, but in practice the management of such accounts and privileges, and connecting to the database with the correct IDs, adds significant complexity
    - Most often a database account is created for the entire web application, with appropriate limitations (e.g., without privileges to create and drop tables)
    - A good compromise is to create database accounts for each class of user or class of operation, so:
      - if Joe is a regular user he wouldn't have delete privileges for the user table
      - Changing user preferences, as an operation type, doesn't require delete privileges
FAQs

- Doesn't SSL protect against this sort of attack?
  - No

- But what if you authenticate users with a username/password over SSL? Then, if the user does SQL injection, the server admins will know who perpetrated the crime, right?
  - Not necessarily; only if you have sufficient audit logging.
Other SQL Injection Methods

- Let's say you've blocked single quotes, double quotes and semi-colons.
- What else can go wrong?
  - How about "\\"?
  - If attacker can inject backslashes, then escaped quotes could get ignored by the database.
PHP-Nuke SQL injection
CVE-2002-1242

- iDefense advisory dated Oct. 31, 2002
- Malicious url:
  - modules.php?name=Your_Account&op=saveuser&uid=2&bio=%5c&EditedMessage=no&pass=xxxxx&vpass=xxxxx&newsletter=,+pass=md5(1)/*
- %5c is the encoding for ‘\’
Let's Look at the SQL

- UPDATE nuke_users
  SET name = '', email = '',
  femail = '', url = 'http://',
  pass = 'xxxxx', bio = '|
  user_avatar = '',
  user_icq = '',
  user_msnm = '',
  newsletter = ,
  pass=md5(1)/*' WHERE uid='2'

- Notice how bio would be set according to the text in red?
  - " (two single quotes) make the database insert a single quote in the field, effectively the same as \'

- Notice how the comment field, /*, is used to comment out the "WHERE" clause for the uid? This means that the query applies to all users!
What Happened?

- All passwords were changed to the value returned by the function "md5(1)"
  - Constant: "c4ca4238a0b923820dcc509a6f75849b"
- Attacker can now login as anyone
A Design Mitigating Database Compromises

- 3-layer Separation of data, code and users
- Scripts (as database users) can only invoke pre-defined queries (code)
Example Using PostGreSQL

- Define 3 database users
  - Table_creator
  - Function_creator
  - Script_user
- PostGreSQL concept: "public" schema
- By default all tables you create belong to the public schema, but you may create other schemas if you wish
  - Schema: A schema is a set of database objects (tables, functions, etc).
Securing the Public Schema

- REVOKE ALL ON SCHEMA public FROM PUBLIC;
- GRANT USAGE ON SCHEMA public TO table_creator;
- GRANT USAGE ON SCHEMA public TO function_creator;
- GRANT USAGE ON SCHEMA public TO script_user;
- GRANT CREATE ON SCHEMA public TO table_creator;
- GRANT CREATE ON SCHEMA public TO function_creator;
- Note that script_user does not get CREATE privileges, and has no privileges on objects created by others
Defining and Securing Tables

- GRANT SELECT, INSERT, UPDATE, DELETE ON users TO function_creator
- Note that function_creator can't alter or drop the table "users"
- Then define functions for the allowed operations.
Defining and Securing Functions

- CREATE FUNCTION set_color(text, text) RETURNS VOID AS $$
  UPDATE users
  SET prefcolor = $1
  WHERE user = $2;
  $$
  LANGUAGE SQL
  EXTERNAL SECURITY DEFINER;

- REVOKE EXECUTE ON FUNCTION set_color(text, text) TO PUBLIC;

- GRANT EXECUTE ON FUNCTION set_color(text, text) TO script_user;
Results

- Script users only have EXECUTE privileges
- The "EXTERNAL SECURITY DEFINER" clause allows the function to execute with the privileges of the function creator
  - This is why the function is not created by the table creator
- Exploits are limited to invoking pre-defined functions
  - Harder to exploit
  - Some things can't be done anymore by attackers
  - This works even if the attacker gets the database password used by the script!
Example: Log database

- Scripts would only be able to read and add new records
- Attacker would be unable to erase activity logs
Closing All SQL Injections: Prepared Statements in Scripts

- `sth = $DBH.prepare("SELECT * FROM set_color(?, ?)")`
  `sth.execute(input_color, 'Joe')`
  `sth.finish`

- No matter what is provided in the input, it can't be used for SQL injection
  - In effect, separate channels are used for code and data

- Conclusion: you can both prevent SQL injection completely and mitigate the consequences of a compromise (e.g., password) with a little work.
Questions or Comments?
About These Slides

- You are free to copy, distribute, display, and perform the work; and to make derivative works, under the following conditions.
  - You must give the original author and other contributors credit
  - The work will be used for personal or non-commercial educational uses only, and not for commercial activities and purposes
  - For any reuse or distribution, you must make clear to others the terms of use for this work
  - Derivative works must retain and be subject to the same conditions, and contain a note identifying the new contributor(s) and date of modification
  - For other uses please contact the Purdue Office of Technology Commercialization.

- Developed thanks to the support of Symantec Corporation
Pascal Meunier
pmeurier@purdue.edu

Contributors:
Jared Robinson, Alan Krassowski, Craig Ozancin, Tim Brown, Wes Higaki, Melissa Dark, Chris Clifton, Gustavo Rodriguez-Rivera