Secure Programming (06-20010)
Chapter 2: General Principles

Christophe Petit

University of Birmingham

Lectures Content (tentative)

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

Secure Programming in a Nutshell

How to write secure programs

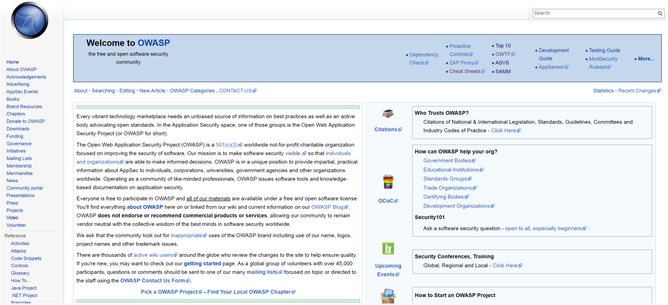
- Follow good recommendations
- Learn general principles
- Get hands-on practice
- Use appropriate tools
- Learn further and stay up-to-date

Follow good recommendations

- Textbooks : Wheeler, Howard-Leblanc, . . .
- Open Web Application Security Project (OWASP)
- Common Weakness Enumeration (CWE)
- Common criteria
- Expert blogs
- Forums
- . . .

OWASP

- Open Web Application Security Project
- Goal : “make software security visible, so that individuals and organizations are able to make informed decisions”



www.owasp.org

OWASP top 10

- Ten most critical web application security risks (2017 draft available online, will be updated in November)
- For each of them : evaluation of exploitability, prevalence, detectability and impact

Threat Agents	Attack Vectors	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	Easy	Widespread	Easy	Severe	App / Business Specific
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

- For each of them : vulnerability assessment checklist, prevention methods, examples and references

OWASP Top 10 (2017 candidates)

T10 OWASP Top 10 Application Security Risks – 2017

A1 – Injection	Injection flaws, such as SQL, OS, XXE, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2 – Broken Authentication and Session Management	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).
A3 – Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A4 – Broken Access Control	Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A5 – Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, platform, etc. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
A6 – Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
A7 – Insufficient Attack Protection	The majority of applications and APIs lack the basic ability to detect, prevent, and respond to both manual and automated attacks. Attack protection goes far beyond basic input validation and involves automatically detecting, logging, responding, and even blocking exploit attempts. Application owners also need to be able to deploy patches quickly to protect against attacks.
A8 – Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. Such an attack allows the attacker to force a victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
A9 – Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10 – Underprotected APIs	Modern applications often involve rich client applications and APIs, such as JavaScript in the browser and mobile apps, that connect to an API of some kind (SOAP/XML, REST/JSON, RPC, GWT, etc.). These APIs are often unprotected and contain numerous vulnerabilities.

A1 Injection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Consider anyone who can send untrusted data to the system, including external users, business partners, other systems, internal users, and administrators.	Attackers send simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, XPath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, expression languages, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. In many cases, it is recommended to avoid the interpreter, or disable it (e.g., XXE), if possible. For SQL calls, use bind variables in all prepared statements and stored procedures, or avoid dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find use of interpreters and trace data flow through the application. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's Java Encoder and similar libraries provide such escaping routines.
3. Positive or "white list" input validation is also recommended, but is not a complete defense as many situations require special characters be allowed. If special characters are required, only approaches (1) and (2) above will make their use safe. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in her browser to send: ' or '1'='1. For example:

http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

References

OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Query Parameterization Cheat Sheet
- OWASP Command Injection Article
- OWASP XXE Prevention Cheat Sheet
- OWASP Testing Guide: Chapter on SQL Injection Testing

External

- CWE Entry 77 on Command Injection
- CWE Entry 89 on SQL Injection
- CWE Entry 564 on Hibernate Injection
- CWE Entry 611 on Improper Restriction of XXE
- CWE Entry 917 on Expression Language Injection

CWE

- CWE = Common Weakness Enumeration
- Maintained by MITRE `cwe.mitre.org/`
- Goals :
  - Classification of common vulnerabilities
  - Baseline to compare software security tools targeting these vulnerabilities
- Developed scoring methodologies, which can be tuned to particular organizations
- See also CVE = Common Vulnerabilities and Exposures, more targeted at products

CWE example : SQL injection

Common Weakness Enumeration
A Community-Developed List of Software Weakness Types

CWE and SANS Institute TOP 25 MOST DANGEROUS SOFTWARE ERRORS

Home > CWE List > CWE- Individual Dictionary Definition (2.11)   ID Lookup: Go

Home | About | CWE List | Scoring | Community | News | Search

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Weakness ID: 89   **Status:** Draft
Abstraction: Base

Presentation Filter: Basic

Description

Description Summary

The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

Extended Description

Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. This can be used to alter query logic to bypass security checks, or to insert additional statements that modify the back-end database, possibly including execution of system commands.

SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Applicable Platforms

Common Consequences

Scope	Effect
Confidentiality	**Technical Impact:** *Read application data* Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.
Access Control	**Technical Impact:** *Bypass protection mechanism* If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
Access Control	**Technical Impact:** *Bypass protection mechanism* If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability.
Integrity	**Technical Impact:** *Modify application data* Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.

Likelihood of Exploit

Very High

Demonstrative Examples

Potential Mitigations

Phase: Architecture and Design

CWE scoring metrics

Group	Name	Summary
Base Finding	Technical Impact (TI)	The potential result that can be produced by the weakness, assuming that the weakness can be successfully reached and exploited.
Base Finding	Acquired Privilege (AP)	The type of privileges that are obtained by an attacker who can successfully exploit the weakness.
Base Finding	Acquired Privilege Layer (AL)	The operational layer to which the attacker gains privileges by successfully exploiting the weakness.
Base Finding	Internal Control Effectiveness (IC)	the ability of the control to render the weakness unable to be exploited by an attacker.
Base Finding	Finding Confidence (FC)	the confidence that the reported issue is a weakness that can be utilized by an attacker
Attack Surface	Required Privilege (RP)	The type of privileges that an attacker must already have in order to reach the code/functionality that contains the weakness.
Attack Surface	Required Privilege Layer (RL)	The operational layer to which the attacker must have privileges in order to attempt to attack the weakness.
Attack Surface	Access Vector (AV)	The channel through which an attacker must communicate to reach the code or functionality that contains the weakness.
Attack Surface	Authentication Strength (AS)	The strength of the authentication routine that protects the code/functionality that contains the weakness.
Attack Surface	Level of Interaction (IN)	the actions that are required by the human victim(s) to enable a successful attack to take place.
Attack Surface	Deployment Scope (SC)	Whether the weakness is present in all deployable instances of the software, or if it is limited to a subset of platforms and/or configurations.
Environmental	Business Impact (BI)	The potential impact to the business or mission if the weakness can be successfully exploited.
Environmental	Likelihood of Discovery (DI)	The likelihood that an attacker can discover the weakness
Environmental	Likelihood of Exploit (EX)	the likelihood that, if the weakness is discovered, an attacker with the required privileges/authentication/access would be able to successfully exploit it.
Environmental	External Control Effectiveness (EC)	the capability of controls or mitigations outside of the software that may render the weakness more difficult for an attacker to reach and/or trigger.
Environmental	Prevalence (P)	How frequently this type of weakness appears in software.

Top vulnerability classes (CWE 2011)

1. Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection")
2. Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")
3. Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")
4. Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting")
5. Missing Authentication for Critical Function
6. Missing Authorization
7. Use of Hard-coded Credentials
8. Missing Encryption of Sensitive Data

Top vulnerability classes (CWE 2011)

9. Unrestricted Upload of File with Dangerous Type
10. Reliance on Untrusted Inputs in a Security Decision
11. Execution with Unnecessary Privileges
12. Cross-Site Request Forgery (CSRF)
13. Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
14. Download of Code Without Integrity Check
15. Incorrect Authorization
16. Inclusion of Functionality from Untrusted Control Sphere
17. Incorrect Permission Assignment for Critical Resource
18. Use of Potentially Dangerous Function

Top vulnerability classes (CWE 2011)

19. Use of a Broken or Risky Cryptographic Algorithm
20. Incorrect Calculation of Buffer Size
21. Improper Restriction of Excessive Authentication Attempts
22. URL Redirection to Untrusted Site ("Open Redirect")
23. Uncontrolled Format String
24. Integer Overflow or Wraparound
25. Use of a One-Way Hash without a Salt

Common Criteria (CC)

- Full name is "Common Criteria for Information Technology Security Evaluation"
- Standard for computer security certification
- Provides assurance to buyers of a security product that specification, implementation and evaluation processes were conducted in a rigorous and standard way

How to write secure programs

- Follow good recommendations
- Learn general principles
- Get hands-on practice
- Use appropriate tools
- Learn further and stay up-to-date

General Principles

- Get your code right
- Check your inputs
- Least privilege and Deny by default
- Secure-friendly architecture
- Defense in Depth

Get your Code Right

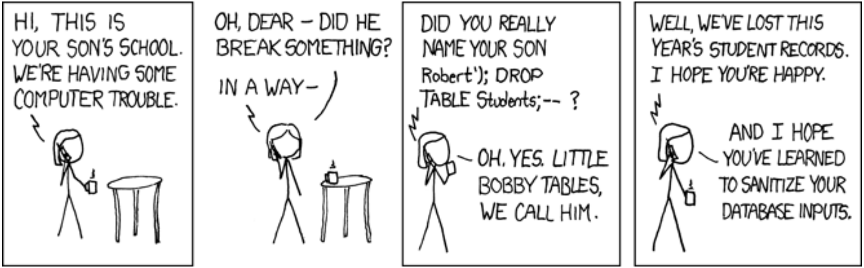
- Consider the following C code

```c
#include<stdio.h>
#include <stdlib.h>
#include<string.h>

int main() {
   int account_balance = 10000;
   printf("Your current balance is %i\n",account_balance);
   printf("How much would you like to withdraw?\n");
   char response[20];
   fgets(response, 20, stdin);
   int withdraw_amount = atoi(response);
   account_balance-=withdraw_amount;
   printf("You have withdrawn %u\n",withdraw_amount);
   printf("Your current balance is %i\n",account_balance);
}
```

- What happens if you withdraw 2,500,000,000 ?
  (for 32-bit integers)

Don't trust external inputs



Picture source : xkcd.com/327/

- Do not mix code and data
- Always assume external outputs/ systems are insecure

Least Privileges

- Give all applications the least privilege they need to work
- Break your applications into small modules, isolate those with highest privileges
- Deny by default - white lists safer than black lists

Keep it Simple

- Start from a simple and clear design
- Break your code into small modules
- Simple code is easier to review
- Simple code is easier to update

Defense in Depth

- Include multiple layers of security
- Block malicious intputs, but still assume some of them might get through
- Deny permissions, and limit damage if they are obtained
- Paranoia is a virtue : plan for worst case
- Fail to secure case
- Least privileges

How to write secure programs

- Follow good recommendations
- Learn general principles
- Get hands-on practice
- Use appropriate tools
- Learn further and stay up-to-date

Get hands-on practice

- Some in this course from the SEED project :
  `http://www.cis.syr.edu/~wedu/seed/`
- Plenty of additional exercices available on the net

How to write secure programs

- Follow good recommendations
- Learn general principles
- Get hands-on practice
- Use appropriate tools
- Learn further and stay up-to-date

Use proper tools

- OS security features
- Secure libraries
- Cryptography
- Static analysis
- Dynamic analysis
- OWASP tools

How to write secure programs

- Follow good recommendations
- Learn general principles
- Get hands-on practice
- Use appropriate tools
- Learn further and stay up-to-date

Learn further and Stay up-to-date

- New vulnerabilities regularly discovered
- New security tools are developed against them
- New applications need to be protected

- Regularly check OWASP, CWE,...
- Plenty of information on the net

Summary

- Get your code right
- Check your inputs
- Least privilege, deny by default
- Secure-friendly architecture
- Defense in Depth
- Stay up-to-date

References

- Howard-Leblanc, Writing Secure Code, Chapter 4
- `cwe.mitre.org/`
- `www.owasp.org`