Security Vulnerabilities Decomposition

Katy Anton
OWASP Top 10
When the report is published
Katy Anton

- Software development background
- Project co-leader for OWASP Top 10 Proactive Controls (@OWASPControls)
- Principle Application Security Consultant
Common Weakness Enumeration

A formal list for software security weaknesses in:
• architecture
• design
• code

Source: https://cwe.mitre.org/
NVD: CWE Categories

Source: https://nvd.nist.gov/vuln/categories/cwe-layout
Injection Category
CWEs in Injection Category

- CWE-77: Command Injection
- CWE-78: XSS
- CWE-91: XML Injection
- CWE-93: CRLF Injection
- CWE-94: Code Injection
- CWE-943: Improper Neutr. of Special El in Query
- CWE-78: OS Cmd Inj
- CWE-78: Argument Inj
- CWE-89: SQL Injection
- CWE-90: LDAP Injection

Source: NVD
Is there another way to look at it?
Decompose the Injection

Data interpreted as Code

Input
- Get / Post Data
- File Uploads
- HTTP Headers
- Database Data
- Config files

Parser
- SQL Parser
- HTML Parser
- XML Parser
- Shell
- LDAP Parser

Output
- SQL
- HTML
- XML
- Bash Script
- LDAP Query

@KatyAnton
Extract **Security Controls**

![Diagram showing the flow of data from Input to Output through Parser]

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Encode Output</th>
<th>Parameterize</th>
<th>Validate Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SQL Injection</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XML Injection</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Code Injection</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LDAP Injection</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Cmd Injection</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Primary Controls**

- Defence in depth
Intrusions

(or lack of Intrusion Detection)
If a pen tester is able to get into a system without being detected, then there is insufficient logging and monitoring in place.
The security control developers can use to log security information during the runtime operation of an application.
The 6 Best Types of Detection Points

Good attack identifiers:
1. Authorisation failures
2. Authentication failures
3. Client-side input validation bypass
4. Whitelist input validation failures
5. Obvious code injection attack
6. High rate of function use
Examples of **Intrusion Detection Points**

**Request Exceptions**

- Application receives GET when expecting POST
- Additional form /URL parameters
Examples of **Intrusion Detection Points**

**Authentication Exceptions**

- Additional variables received during an authentication like ‘admin=true’
- Providing only one of the credentials
  
  The user submits POST request which only contains the username variable. The password was removed.
Examples of Intrusion Detection Points

**Input Exceptions**

- Input validation failure on server despite client side validation
- Input validation failure on server side on non-user editable parameters
  - e.g.: hidden fields, checkboxes, radio buttons, etc
Secure Data Handling: Basic Workflow

- Application Server
- Operating System
- Software Application
- Log Exceptions
- Param Queries
- Validate Data
- Encode output

Diagram flow:
- Data from DB to Param Queries
- Param Queries to Validate Data
- Validate Data to Encode Output
- Encode Output to Log Exceptions
- Log Exceptions to Application Server
Sensitive Date Exposure

Data at Rest and in Transit
## Data

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Encryption</th>
<th>Hashing</th>
</tr>
</thead>
</table>
| Data at **Rest**: Requires initial value  
  E.q: credit card         |            | ✓       |
| Data at **Rest**: Doesn’t require initial value  
  E.q: user passwords      |            | ✓       |
| Data in **Transit**         |            | ✓       |
Data at Rest: Design Vulnerability example

How Not to Do it!

In the same folder - 2 file:

encrypted-password.txt
password-entities.txt

The content of password.txt:

cryptography.seed=abcd
cryptography.salt=12345
cryptography.iterations=1000

cipher_key = PBKF2(password, salt, iterations, key_length);
Encryption: Security Controls

Strong Encryption Algorithm: AES

Key Management

• Store unencrypted keys away from the encrypted data.
• Protect keys in a Key Vault (Hashicorp Vault / Amazon KMS)
• Keep away from home grown key management solutions.
• Define a key lifecycle.
• Build support for changing algorithms and keys when needed
• Document procedures for managing keys through the lifecycle

Source: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html
Data in Transit: Security Controls

Diagram showing data flow with TLS encryption between various components:
- Laptop to Application Server
- Application Server to Operating System
- Operating System to Software Application
- Software Application to Database
- Database to Application Server
- Application Server to Operating System
- Operating System to Laptop

Key: TLS for encryption.
Third Party Components

Using Software Components with Known Vulnerabilities
Apps with at least 1 vulnerable component:

- 85.7% of .Net applications
- 92% of C++ applications

Root Cause

- Difficult to understand
- Easy to break
- Difficult to test
- Difficult to upgrade
- Increase technical debt
What is **Attack Surface**?

Sum of the total different points through which a malicious actor can try to **enter data into** or **extract data from** an environment.
Fundamental Security Principle

Minimize the attack surface area
Components Examples

Example of external components:
• **Open source** libraries - for example: a logging library
• **APIs** - for example: vendor APIs
• **Packages** by another team within same company
Example 1: Implement Logging Library

- Third-party - provides logging levels:
  - FATAL, ERROR, WARN, INFO, DEBUG.

- We need only:
  - DEBUG, WARN, INFO.
Simple Wrapper

Helps to:
• Expose only the functionality required.
• Hide unwanted behaviour.
• Reduce the attack surface area.
• Update or replace libraries.
• Reduce the technical debt.
Example 2: Implement a **Payment Gateway**

Scenario:

- Vendor APIs - like payment gateways
- Can have more than payment gateway one in application
- Require to be inter-changed
Adapter Design Pattern

- **Converts** from provided interface to the required interface.
- A single Adapter interface can work with **many** Adaptees.
- **Easy** to maintain.
Example 3: Implement a **Single Sign-On**

- Libraries / packages created by another team within same company
- Re-used by multiple applications
- Common practice in large companies
Façade Design Pattern

- Simplifies the interaction with a complex sub-system
- Make easier to use a poorly designed API
- It can hide away the details from the client.
- Reduces dependencies on the outside code.
Secure Software Starts from Design!

**Wrapper**
To expose only required functionality and hide unwanted behaviour.

**Adapter Pattern**
To convert from the required interface to provided interface

**Façade Pattern**
To simplify the interaction with a complex sub-system.
How often?
Rick Rescorla

- United States Army office of British origin
- Born in Hayle, Cornwall, UK
- Director of Security for Morgan Stanley at WTC
Security Controls Recap
Security Controls In Development Cycle

- Application Server
- Logs: Log Exception
- Operating System
- OS Command: Param Data, Secure Date
- Software Application: Encapsulation
- Database: Param Queries, Validate Input
- Key Management: Encapsulate Secure Data

Final Takeaways

Focus on Security Controls

CWEs

Incorporate prevent, verify, early and often

CWEs

Verify, early and often
References

• OWASP Top 10 Proactive Controls
  https://owasp.org/www-project-proactive-controls/

• OWASP Cheat Series
  https://cheatsheetseries.owasp.org/
Thank you very much