Building Secure Software

Example

Registration Form

User id: Required and must be of length 5 to 12.

Password: Required and must be of length 7 to 12.

Name: Required and alphabates only.

Address: Optional.

Country: (Please select a country) Required. Must select a country.

ZIP Code: Required. Must be numeric only.

Email: Required. Must be a valid email.

Sex: Male Female Required.

Language: English Non English Required.

About: Optional.

Submit

http://www.w3resource.com/javascript/form/javascript-sample-registration-form-validation.php

Building Secure Software

Why (1/3)?

Traditionally, software development concentrated on building a system which works to specifications, performs well under load, reliably produces the correct output for the given input, and is maintainable.

A secure software development process tries to integrate security best practices and sets up metrics and governance around it.

Software security is the activity of trying to make software behave (well) under intentional malicious attack.

Building Secure Software

Why (2/3)?

Building software that behaves well under normal circumstances is a challenge in itself, but building it to behave when a malicious attack happens requires additional skills like these:

- Think like an attacker, i.e. identify possible attack vectors and scenarios
- Expect the unexpected, e.g. supplying a username which is 1MB in length, inputting non-printable characters, or adding URL or SQL parameters
- Be able to detect possible attacks and find ways to defend against them
- …

Building Secure Software

Why (3/3)?

Managing a software development process requires paying attention to security, including making everyone responsible for security (just like you do for quality, deadlines, etc.). Depending on the organization and management style, either use a "top-down" approach to security or create a "risk portfolio" of your applications.

- The "top-down" approach involves setting goals and providing tools (incl. training) for the software development community right from the start and whenever needed.
- The "risk portfolio" approach involves taking stock of your application (new or existing) and defining which ones are the most critical or valuable ones for your organization – and then provide the necessary security tools for these applications.

Building Secure Software

Below is the top 5 of the SANS/Mitre list of most common programming errors (for more details see the reference listed at the end):

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function

Building Secure Software

"list of recent vulnerabilities for which exploits are available" from May 8, 2014

- Title: Adobe Flash Player Integer Underflow Remote Code Execution

  Description: Integer Underflow in Adobe Flash Player before 11.7.700.261 and 11.8.x through 12.0.x before 12.0.0.44 on Windows and Mac OS X, and before 11.2.202.336 on Linux, allows remote attackers to execute arbitrary code via unspecified vectors

- Title: OpenSSL TLS Heartbeat Extension Buffer Oveflow Information Disclosure Vulnerability (Heartbleed)

  Description: The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c.

- Title: Microsoft Internet Explorer Use-after-Free Vulnerability

  Description: Use-after-free vulnerability in VGX.DLL in Microsoft Internet Explorer 6 through 11 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, as exploited in the wild in April 2014.

Building Secure Software

Best practices and real-life examples

Using references like the SANS/Mitre Top 25 programming errors or the OWASP Top 10 is a great starting point for learning what an attacker uses. These lists also allow software developers to determine what they need to defend against – and even how to test whether their software does so successfully.

Insecure software is caused by flaws or bugs any of these seven plus one areas – Gary McGraw calls them "seven plus one kingdoms":

- *Input Validation and Representation*
- *API Abuse*
- *Security Features*
- *Time and State*
- *Errors*
- *Code Quality*
- *Encapsulation*
- *Environment*

Building Secure Software

Input Validation and Representation

See also : http://www.hpenterprisesecurity.com/vulncat/en/vulncat/IPV.html

- Copying data into a buffer which exceed the buffer's size and the use of “format strings” in an input or incorrect string termination can lead to memory corruption and executing of unwanted code
- Path traversal or commands injected from outside input (human or not) or the environment can lead to execution of malicious commands, e.g. “rm -rf”, “rmdir *.* /s /q”, or “c:\cmd.com”
- Cross-site scripting can lead to a browser executing malicious scripts, e.g. Active X or Java scripts
- SQL injection can lead to data leakage and up to database corruption
- Allowing un-validated data in an HTTP header allows injection of malicious code, e.g. redirects
- Illegal pointers, integer overflows or undefined variables can lead to random code execution
- Log forging can allow an attacker to introduce malicious content into logs which can lead to execution of malicious commands down the road
- Loading libraries or executing commands from untrusted sources can lead to execution of malicious commands
- Uncontrolled manipulation of settings can lead to the system misbehaving
- Unused, outdated or disabled data validation (e.g. from web forms, URL parameters) can lead to memory corruption and executing of unwanted code

Building Secure Software

API Abuse

See also : http://www.hpenterprisesecurity.com/vulncat/en/vulncat/APIA.html

- Dangerous functions – those that cannot be safely executed – should never be used, e.g. “gets”
- Improper setup of “chroot” jails or use of the “chroot” system call can lead to executing code or commands outside the secure area
- Using sockets in web applications are error prone
- Unhandled or undetected exceptions can lead to program crashes, e.g. allow Denial-of-Service attacks
- Using privileges that are not really needed can increase existing risk, e.g. a process running as “superuser” has access to all resources and settings, where it might be sufficient to run as “joe”
- Unchecked return values can lead to unexpected behavior and can even lead to program crashes

Building Secure Software

Security Features and Time and State

See also : http://www.hpenterprisesecurity.com/vulncat/en/vulncat/SF.html, http://www.hpenterprisesecurity.com/vulncat/en/vulncat/TnS.html

- Using (badly implemented) pseudo-random number generators can lead to insufficient or predictable entropies which compromise encryption
- Use of elevated privileges can increase existing risk
- Access controls which are not used consistently across all components of a system can lead to circumvention of these controls
- Empty, hard-coded, easy-to-guess passwords, passwords stored in plain text or in (insecure) configuration files can lead to system compromise by unauthorized users
- Weak encryption does not protect data (incl. Passwords) adequately
- Mishandling private information compromises user privacy and can lead to criminal and civil prosecution

- Invalid or not properly synchronized clocks can lead to encryption, synchronization and authentication problems, e.g. for VPN access
- Deadlocks or signal race conditions can lead to unexpected behavior or crash
- Insecure file creation (esp. for temporary files) leaves the system vulnerable to data leakage or manipulation
- Using threads in web applications is highly error prone and should be avoided

Building Secure Software

Errors and Code Quality

See also : http://www.hpenterprisesecurity.com/vulncat/en/vulncat/ERR.html, http://www.hpenterprisesecurity.com/vulncat/en/vulncat/CQ.html

- Not detecting and handling errors (e.g. in a “catch” block) can lead to program crashes, e.g. allow Denial-of-Service attacks
- Overly broad error handling introduces more complexity and therefore increases security risks
- Unchecked return values can lead to unexpected behavior and can even lead to program crashes

- Coding errors like “double free” a pointer, “null dereference”, resources/variables which are never freed after use – or used after being released, use of uninitialized variables and failure to release a system resource can lead to unexpected behavior and serious problems like buffer overflow, null pointer exceptions or program crashes
- Inconsistent implementations across several platforms/operating systems cause portability problems and different user experience
- Using obsolete, deprecated or insecure function indicate neglected code

Building Secure Software

Encapsulation

See also : http://www.hpenterprisesecurity.com/vulncat/en/vulncat/Encap.html

- Data leakage can occur between user sessions (e.g. via servlets or shared objects), leftover debug code, or allowing private data to be changed through public methods or by making them publicly available
- Debug statements which contain internal information such as system environment, database organization etc. – allow an attacker to learn more about the software and adjust attack plans accordingly
- Comparing classes by name without checking the contents can lead to false assumptions and implementations
- Mixing trusted and untrusted data in the same structure (e.g. a database table) lead to mistake all data in the structure as being trusted

Building Secure Software

Environment

See also : http://www.hpenterprisesecurity.com/vulncat/en/vulncat/Env.html

- NEVER hard-code passwords in code, configuration files or databases
- Creating binaries which contain debug statements or lack custom error handling can lead to data leakage which allow an attacker to learn more about the software and adjust attack plans accordingly
- Creating binaries using insecure compiler options can compromise security
- Misconfiguration of encryption (e.g. weak cipher, insecure algorithm, small entropy pool, keys and IDs smaller than 128bit) can lead to a false sense of security
- Weak access permissions to objects can lead to lax security

Building Secure Software

References (1/2)

But all this is moot if it requires development of specific security code in each project and even subroutine or function. Thankfully there are several tools which aide in designing and implementing secure code:

- OWASP Cheat Sheets (very helpful in designs): https://www.owasp.org/index.php/Cheat_Sheets
- OWASP Enterprise Security API: https://www.owasp.org/index.php/ESAPI
- Gary McGraw Podcast: https://www.cert.org/podcast/transcripts/20080820mcgraw-transcript.pdf
- US-Cert "Build Security In" website: https://buildsecurityin.us-cert.gov
- HP Fortify Taxonomy: http://www.hpenterprisesecurity.com/vulncat/en/vulncat/index.html

Building Secure Software

References (2/2)

- SANS/Mitre Top 25 programming errors: http://cwe.mitre.org/top25
- OWASP Top10 attacks and mobile risks: https://www.owasp.org/index.php/Top_10_2013-Top_10, https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks
- SANS Newsletter @RISK: The Consensus Security Vulnerability Alert: Vol. 14, Num. 18
- A free self-contained training environment for Web Application Security penetration testing: https://www.mavensecurity.com/web_security_dojo
- OWASP "WebGoat" project: https://github.com/WebGoat/WebGoat
- A repository of all HTML5 Security resources: http://www.andlabs.org/html5.html