Appunti Security Testing

Diego Oniarti

Anno 2024-2025

Contents

1 Heap vs Stack

Ciò che allochi sulla heap non viene deallocato quando la funzione finisce. Quello che viene allocato dalla stack viene distrutto a fine lezione. Questo lo ha detto anche Roveri nel primo semestre di triennio.

2 Classification and groups of bugs / errors

There are many ways of dividing errors into taxonomies. The major ones consist in ranking them on a certain metric. This can be:

- **severity:** how much of an impact the bug can have on the system
- **priority:** how quickly the issue should be resolved, based on the business impact and the prihect imeline.
- **nature:** it refers to technical characteristics of the bug/error (es: functional or security, design or code specific, etc..)

Some of these chriteria are objective and measurable while some other are more subective and require some informed evaluation.

2.1 Why classify bugs?

The classification can be the first step of investiation for:

- conducinf an **effective** (more targetted) testing acrivity
- improving the development process.
  Some development techniques are more suited for different kinds of task
- driving bug-fixing
  By knowing the common vulnerabilities of my kind of application I can test those more thoroughly. Like testing SQL injections when we write a webapp.

3 Injection

An application if vulnerable to attack when user supplied data is not validated, filtered, or sanitized.

4 SQL injection

4.1 Error-based SQL injection

Un tipo di injection che mira a causare un errore serverside. L'attaccante piò usare l'errore per estrapolare informazioni utili riguardo il sistema.

4.2 Union-based SQL injection

Injection che compie delle union SQL per ottenere più dati di quanti sarebbero normalmente accessibili all'utente.

4.3 Boolean-based Blind SQL injection

Non si ottiene un messaggio dal server ma si mandano query contenenti operatori booleani per inferire informazioni sulle tabelle.

4.4 Prevenzione

Usa i prepared-statement.

5 Cross-Site Scripting (XSS)

Attacchi XSS consistono nell'iniezione di codice "maliziono" in siti fidati.
Questo è un tipo di attacco *client side*. Quindi l'attaccante ha accesso a dati client side come cookie, token di sessione, etc.. Questo rende l'attacco meno dannoso di una SQL-injection. Ma piò comunque essere molto pericoloso.

Ci sono molti tag html che possono portare all'esecuzione di codice javascript. Per esempio: script, body, img, ifreame, input, etc..

Possibili soluzioni

- Non fidarsi mai degli input utente
- Disattivare i cookie con la flag "htmlonly" se il sito non ne necessita

6 Command Injection

Command Injection è un tipo di attacco in cui un programma esegue dei comandi. Un utente malintenzionato può manipolare il programma in maniera che i comandi eseguiti siano dannosi o forniscano informazioni.
Questo è un tipo di attacco pericoloso in quanto l'esecuzione arbitraria di comandi piò portare a danni ingenti alla macchina o perdita di dati importanti.

Requisiti per l'attacco

- The app shold have privileges/permissions to execute system commands
- The app should use user-provided data as part of system commands
- The user-procided data should not be escaped/sanitized before use

Mitigazione

- Non usare funzioni di esecuzione shell/SO
- Non usare input utente nei comandi shell/SO
- Sanatizzare l'input
  - Whitelist di caratteri/keyword non pericolosi
  - Blacklist di caratteri/keyword pericolosi
  - Escaping dell'input
- Parapetrizzare i comandi (simile a prepared statements)
- Principio di *Least Provilege*

Bisogna anche scegliere saggiamente il metodo in cui il programma invia i comandi al sistema operativo. Ad esempio in C c'è il comando `execvp` che è più sicuro di `system`. In Java ci sono `Runtime.exec` e il più sicuro `ProcessBuilder`.

7 File Inclusion Attack

Un attacco di file inclusion mira a programmi che fanno utilizzo di dati salvati su file a runtime. Se l'attaccante può modificare i file (o aggiungerne di nuovi) prima che vengano inclusi può affliggere il programma.

Questo tipo di attacco si suddivide in *Local File Inclusion (LFI)* e *Remote File Inclusion (RFI)*. Il principale rischio nel primo caso è la trapelazione di informazioni (esponendo file locali all'attaccante), mentre nel secondo caso è più probabile che l'attaccante carichi file dannosi verso il sistema.

7.1 File access: directory traversal

L'attaccante usa una serie di "*../*" per esplorare il file system del sistema target. Invece che sanatizzare l'input, questi casi possono essere affrontati configurando permessi adeguati all'applicazione.

8 Error Handling

L'error handling è un meccanismo usato per risolvere o gestire errori che si verificano durante l'esecuzione di un pogramma. La gestione degli errori è una parte integrante della sicurezza di un sistema.

Un attaccante inizia dalla fase di ricognizione, in cui deve ottenere informazioni riguardo il sistema che sta attaccando. Queste informazioni possono essere fatte trapelare da messaggi d'errore, stack trace, e altre forme di error handling.

Anche il modo in cui viene implementato il codice piò portare a far trapelare informazioni. Un errore che porta il sistema a crashare può fornire all'attaccante un'intera stack trace dell'errore che include altri dati.

Mitigazione Vulnerabilità di questo tipo possono essere mitigate filtrando i messaggi di errore che vengono mandati all'utente o gestendo i casi d'errore in modo che non vengano sollevate exception / crash.

9 Insecure Direct Object Reference (IDOR)

IDOR is a type of access control vulnerability that arises when an application uses usersupplied input to access objects directly. A direct object reference occurs when a developer exposes a reference to internal implementation objects (e.g., files, directories, database keys, session ids, query parameters) without appropriate validation mechanisms, thus allowing attackers to manipulate these references to access unauthorized data.

10 Client-Side Validation

La validazione di input dal lato del client è utile per alleviare lo stress sul server (evitando un avanti e indietro), ma i controlli vanno sempre replicati sul server alla fine.

11 Client-Side Filtering

Certe volte il server manda più informazioni del dovuto al client, fidandosi che sia quest'ultimo a filtrare queste informazioni prima di mostrarle all'utente. Questa è una bad practice per ovvi motivi.

12 Phases of testing

12.1 Verification vs Validation

Verification asks the question "are you building it right?" while *Validation* checks "Are you building the right thing".
We do both to ensure that the system does the right thing and that it does so safely

12.2 Static vs Dynamic Analysis

L'analisi statica viene svolta sul codice effettivo, mantre quella dinamica è svolta sul processo in esecuzione.

12.3 Test cases - dynamic

Un test case descrive una procedura che mette alla prova il sistema

Oracle "l'oracolo" è l'output atteso del programma nel caso il test case vada a buon fine. Deve essere stabilito manualmente dallo sviluppatore.

12.4 Test cases - static

???

13 Tainted Variable

Una variabile è detta *"tainted"* quando non è checkata.
Quando il vaore di una tainted variable è checkato, diventa *untainted*.

L'analisi di queste variabili nel codice è detta *"taint analysis"*, è può essere di tipo statico o dinamico.

14 Testing

A *Test Suite* is a set of test cases. When defining a test suit there are some generic problems:

1. How to identify the required test inputs
   It is not possible to test EVERY input. One approach is to test with random inputs, but various approaches depend on the type of problem.
2. How to identify the right test oracle to be used
3. When to stop testing
   Testing could end when some coverage criteria is met, or when some resource (like time) runs out. There is also Mutation testing.

14.1 test coverage

Test coverage aims at ensuring that a test suite is comprehensive enough and that all relevant and critical applications aspects and functionalities are covered. It is based on "coverage measures or criteria"

14.2 integration testing

14.3 test coverage criteria

There are many criteria that can be used to valuate test coverage. Some of these are

- Code coverage
- data flow code coverage
- Boundary value coverage
- risk coverage
- requirements coverage
- type of bugs

But there are many more.

Different criteria are better suited for different kinds of applications. Code coverage is the easiest to use (especially if the code can be easily instrumented) but it gives little guarantee about the security of the code.

14.4 Statement Coverage

Statement coverage aims at involving the execution of all the executable statements at least once.

Vedi slide 14 per l'esempio con la tabellina fatta bene.

Un test case è detto *ridondante* se testa degli statement che sono già testati da altri casi.
Un test che è completamente ridondante può essere eliminato dalle test suit.

14.5 Decision Coverage

Decision coverage reports the true or false outcomes of each boolean expression (or code decision point)

14.6 Condition Coverage

Condition coverage reveals how the (atomic) logical operands in the conditional statement are evaluated. This is similar to the decision coverage, but instead of evaluating the 'IF' statement as either true or false, we do the evaluation on the atomic parts of the boolean conditions.

14.7 Branch coverage.

Branch coverage tests every outcome from the code to ensure that every branch (CFG edge) es executed at least once

14.8 Code coverage testing tools

- Emma: java eclipse plugin
- JacCoCo: Specializzato per le servlet

15 mutation testing

Una tecnica usata per valutare la qualità di una test suite. Introduciamo un bug noto nel programma e verifichiamo che la test suite rilevi questo bug.
In caso contrario c'è una mancanza nella suite.

Alcune assunzioni sono:

- solo un cambiamento è introdotto alla volta
- l'obbiettivo può essere raggiunto misurando *quanto bene trova il bug introdotto*
- il bug introdotto è indicativo delle problematiche effettive che potrebbero essere nel codice.
- errori veri e introdotti artificialmente non devono essere identici, ma le differenze non dovrebbero affliggere l'obbiettivo.

15.1 Mutanti

- Un **Mutant** è il programma modificato
- Un mutant è **Killed** quando è rilevato dalla test suite
- Un **Mutant Score** è $\frac{killedmutants}{totalmutants} * 100$
- Una test suite è **mutation-adequate** se il mutant score è 100

15.2 Common mutation

Ci sono dei mutant che vengono comunemente usati

operand replacement Cambiare operatori o operandi in una espressione. $(x > y)$ può diventare $(x < y)$ o $(x > 3)$

Rimuovere l'*else* in un blocco condizionale, rimuovere l'intero *if-else*, aggiungere un return

Queste modifiche non devono causare errori a compile time ma solo runtime (deve compilare).

Altri mutanti comuni:

- remove regex sanitization
- remove path traversal sanitization
- remove http-only flag from cookie
- permit sql injection
- sql
  - add or/and clauses
  - add ";"
  - change the encoding of whitespaces

15.3 tools

Two tools for mutation testing are *stryker mutator* e *PITest (eclipse plugin)*

16 fuzzing

Fuzzing è la generazione automatica di input con l'obbiettivo di crashare l'applicazione.

Gli input sono "casuali" ma tipicamente sono stringhe molto lunghe, numeri grandi, input "strani" in generale, e altri input che possano metter alla prova l'applicazione.

Questo include numeri ai limiti dei range rappresentabili, stringhe composte da caratteri particolari, etc..

L'oracolo di un fuzzer quindi è un crash.

Tecniche più sofisticate non mirano a un crash ma a altre metriche come memory leak, stati non validi dell'applicazione, accessi invalidi a memoria, etc.. Questi test sono molto più difficili da progettare.

Il fuzzing ha il vantaggio di essere molto semplice, in quanto ci sono tools che possono dare molti input automaticamente a un app, ma non è molto efficace per alcuni tipi di applicazione.

17 Blackbox vs graybox

18 Esecuzione simbolica

Esecuzione del software non con valori effettivi ma con simboli che rappresentano la classe di tutti i valori che i simboli potrebbero avere. Poi vengono generati dei *path constraint*, delle limitazioni e relazioni tra i simboli che sono valide globalmente o in determinati branch di codice.

La path condition ci dice quale relazione deve essere presente tra gli input per far si che il programma segua un certo path.

Possiamo definire degli input validi e tenere tracci della esecuzione simbolica durante quella effettiva. La condition ci permette poi di definire nuovi input che non matchino la path condition e ripetere il processo.

Durante la symbolic execution potremmo imbatterci in chiamate a funzioni private. In questo caso il simbolo prende il valore di ritorno della funzione.