Università degli Studi di Padova

Dipartimento di Diritto Pubblico, Internazionale e Comunitario
Dipartimento di Matematica "Tullio Levi-Cita"

Corso di laurea in Diritto e Tecnologia
a.a. 2023/2024

Tassonomia dei software bug

Relatore
Prof. Silvia Crafa

Studente
Pirdeni Mattia

Sommario

Questa tesi esplora la tassonomia dei bug software, analizzando diverse classificazioni e metodologie per categorizzare gli errori nel codice. Vengono presentate tassonomie generali e specifiche per diversi tipi di software, evidenziando l'importanza di una classificazione sistematica per migliorare lo sviluppo e la manutenzione del software. La tesi esamina inoltre alcuni casi storici significativi di bug, come Log4Shell, Y2K e gli incidenti del Boeing 737 MAX, analizzandone cause e conseguenze. Infine, viene discusso il Cyber Resilience Act dell'Unione Europea e il ruolo che una tassonomia dei bug può svolgere nell'applicazione di questa normativa.

Indice

1 Introduzione **1**
1.1 Definizioni dei termini utilizzati per parlare di bug 1
1.2 Origini del termine "bug" 2
1.3 Il Costo dei Bug 3

2 Tassonomia dei bug **5**
2.1 Come classificare un bug 5
2.2 Tassonomia dei bug 5
2.3 Esempi di tassonomie 6
2.3.1 Tassonomia di Boris Beizer 6
2.3.2 Tassonomia bug dei videogiochi 8
2.3.3 Tassonomia dei bug dei sistemi di deep learning 10
2.3.4 Tassonomia vulnerabilità 11
2.3.5 Tassonomia per la classificazione automatica dei bug 12

3 Bug significativi **15**
3.1 Log4Shell 15
3.2 Y2K e Y38K 18
3.3 Knight Capital Group 20
3.4 Therac-25 22
3.5 Boeing 737 MAX 25

4 Aspetti legali **29**
4.1 Cyber Resilience Act 29

Bibliografia **33**

Elenco delle figure

1.1 Falena nel diario 2
1.2 Costo dei software bug 3

2.1 Tassonomia bug videogiochi 8
2.2 Tassonomia vulnerabilità 11

3.1 Exploit di Log4Shell 16
3.2 Prima dell'overflow 20
3.3 Dopo l'overflow 20
3.4 Boeing 737 MAX 26
3.5 Sistema anti-stallo del Boeing 737 MAX 27

Capitolo 1

Introduzione

1.1 Definizioni dei termini utilizzati per parlare di bug

Come primo argomento di questa tesi, è opportuno definire chiaramente i termini che vengono utilizzati per parlare in generale dei bug del software. Sono diversi i termini che vengono utilizzati in questo contesto, spesso anche in modo intercambiabile. Essi possono avere significati diversi a seconda del contesto in cui vengono utilizzati, oppure essere quasi sinonimi. I termini principali che vengono utilizzati per riferirsi ai bug sono[1]:

- Bug: il termine bug viene utilizzato per definire un comportamento del software che non è conforme alle specifiche o alle aspettative dell'utente. Un bug può derivare da un qualsiasi errore durante la fase di sviluppo del software, come ad esempio un errore di logica, ma vengono inclusi ad esempio anche errori di battitura.

- Errore: con errore si intende una situazione in cui il programmatore commette un errore nella scrittura del codice, cioè esso non comprende correttamente le specifiche del software oppure si sbaglia nel codificare la soluzione nel codice, e ciò causa potenzialmente la scrittura di codice contente bug.

- Glitch: è un termine inglese che significa un "piccolo problema o difetto che impedisce a qualcosa di funzionare correttamente" [2]. Il termine si riferisce ad un bug solitamente non grave e che si manifesta soltanto in determinate situazioni.

- Vulnerabilità: è un difetto nel codice che può essere sfruttato da un attaccante per compromettere la sicurezza del sistema.

1.2 Origini del termine "bug"

Diverse fonti[1] citano come primo utilizzo del termine bug per riferirsi ad un problema di un software il caso descritto nel diario di Grace Hopper dove si racconta che un computer, il Mark II ad Harvard, smise di funzionare per colpa di una falena che si incastrò all'interno della macchina, causando un malfunzionamento. Quel momento fu descritto come "il primo caso di un vero "bug" trovato".

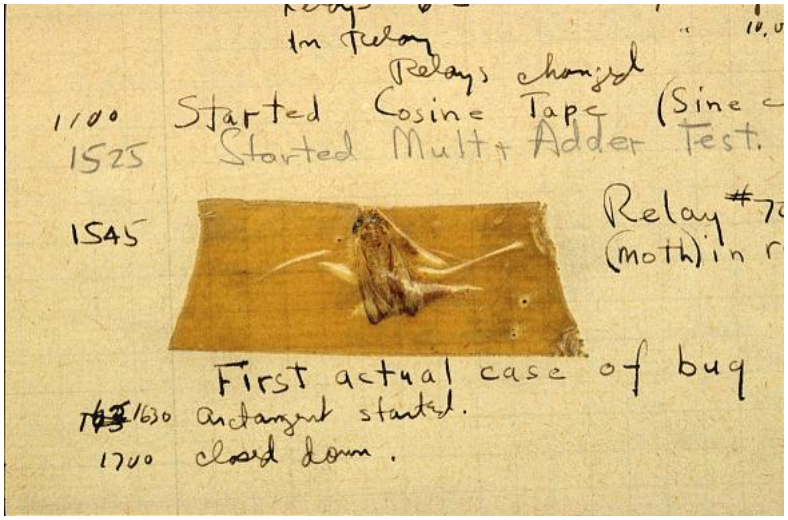

Figura 1.1: Falena nel diario
[3]

In realtà però questa spiegazione ha diverse imprecisioni (ad esempio il diario probabilmente non era di Hopper), in quanto è possibile trovare riferimento al termine "bug" nel *Oxford English Dictionary (OED)* già dal 1875 con il significato di "difetto o guasto in una macchina (sopratutto elettronica) o in un processo." [4]; inoltre sempre nel *OED* è possibile trovare una citazione ad un articolo del Paul Mall Gazette[2] dove si scrive si racconta che Thomas Edison usò il termine "bug" per riferirsi ad un problema riguardante il suo fonografo. Nonostante ciò l'evento della falena nel Mark II ha significativamente aiutato a diffondere l'utilizzo di "bug" per riferirsi ad un problema sia hardware che software. [5]

[1] Ad esempio The Naked Computer (1983) di Jack B. Rochester e John Grant'z, New York Times (1984, 1985), Times (1984)

[2] Un quotidiano inglese fondato a Londra nel 1965

1.3 Il Costo dei Bug

Secondo il CISQ[3], il software di bassa qualità è costato all'economia statunitense 2,08 bilioni di dollari solo nel 2020. I costi dei bug software non si limitano ai costi diretti che uno sviluppatore deve sostenere per correggerli, ma sono da considerare anche i tempi di inattività dei lavoratori, le interruzioni e i ritardi. Possono seguire anche perdite di reputazione, dato che un software difettoso può indicare ai clienti un azienda che produce software di bassa qualità. Inoltre, i bug possono introdurre rischi per la sicurezza, con un impatto finanziario sotto forma di attacchi informatici, violazioni dei dati e furti finanziari.

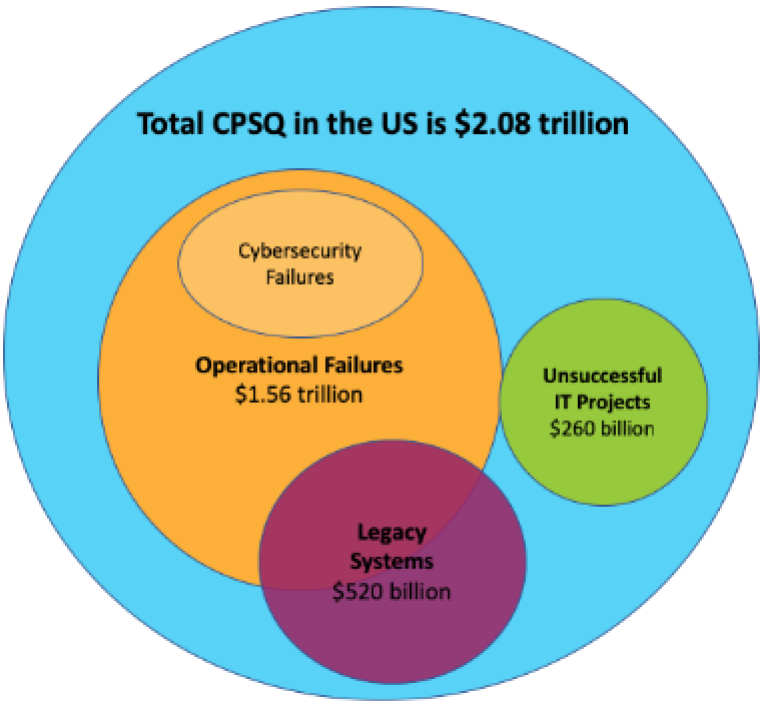

Figura 1.2: Costo dei software bug
[6]

Il ciclo di vita dello sviluppo del software (SDLC) è costituito dai passaggi standardizzati che un sviluppatore attraversa quando crea e mantiene il software. Questi passaggi includono pianificazione, analisi, progettazione, implementazione, distribuzione e manutenzione.

I bug possono verificarsi in qualsiasi fase dell'SDLC. Se un bug viene introdotto nelle prime fasi dell'SDLC e non viene immediatamente risolto, i suoi costi aumenteranno solo man mano che progredisce attraverso l'SDLC. Ad esempio, correggere un bug nella fase di pianificazione può costare €100 se trovato presto, ma lo stesso bug può diventare un

[3]Consortium for Information & Software Quality

problema da €10.000 se scoperto più tardi nella fase di produzione. Questo perché un bug può avere un effetto domino che porta a costi e ritardi aggiuntivi.

Trovare il giusto equilibrio tra la correzione dei bug e lo sviluppo di nuovi miglioramenti è una parte importante dello sviluppo del software. Uno sviluppatore deve trovare il giusto equilibrio che meglio si adatta a loro. Un rapporto tipico è spendere il 20% del proprio tempo su compiti reattivi come il debugging e l'80% del proprio tempo su compiti proattivi come la creazione di nuove funzionalità del prodotto. Trascorrere più del 20% del tempo sul debugging può indicare che ci sono problemi nel processo di sviluppo che stanno influenzando la loro produttività complessiva.

È comunque importante spendere quel 20% di tempo utilizzato per il debugging, nella maniera migliore, identificando precocemente i bug, dato che se questi ultimi viene rilevati nelle prime fasi dello sviluppo i tempi di identificazione e risoluzione saranno sostanzialmente ridotti.[7]

Capitolo 2

Tassonomia dei bug

2.1 Come classificare un bug

Un bug può essere classificato in molti modi differenti a seconda di quale elemento specifico si vuole mettere in evidenza. Esempi di classificazioni comuni sono:

- **Causa**: classificazione in base alla cause che lo ha generato, come ad esempio un errore di logica o di sintassi.
- **Effetto**: classificazione in base a come il bug si manifesta nell'applicazione, ad esempio crash, problemi di performance, glitch grafici.
- **Gravità**: classificazione basata sull'impatto che ha il bug sull'applicazione, ad esempio un bug che rende difficile leggere un testo è meno grave di un bug che causa un crash.
- **Priorità**: classificazione basata sull'urgenza di risolvere il bug, spesso basato sulla gravità di esso.

2.2 Tassonomia dei bug

La tassonomia dei bug è una classificazione dei bug in modo gerarchico che può includere la cause, effetti o qualsiasi altro aspetto del bug.

Data l'enorme diversità dei software che possono essere creati non è possibile creare una tassonomia unica che raggruppi tutte le possibili categorizzazioni, ma è necessario scegliere uno o piú criteri di classificazioni. Infatti una tassonomia deve essere sviluppata tenendo in conto le specifiche del software a cui si riferisce, come ad esempio il linguaggio utilizzato,

il tipo di applicazione o l'ambiente di esecuzione, inoltre esse deve essere dinamica ad adattarsi alle nuove problematiche che possono emergere.

Classificare i bug risulta molto utile per diversi motivi, tra cui:

- Aiuta a decidere a quali bug dare priorità.
- Permette di assegnare il bug alla figura piú adeguata per la risoluzione.
- Fornisce una panoramica generale delle tipologie di bug presenti nel software e permette di individuare eventuali punti critici.
- Aiuta i nuovi sviluppatori a trovare punti di partenza per la risoluzione dei bug.
- Permette di stabilire punti specifichi sui quali scrivere test.
- Fornisce ai software tester una guida su come testare il software.

Di seguito verranno illustrate alcune tassonomie che si adattano a diversi tipi di software, alcune sviluppate per una classificazione generale di alto livello, altre invece più specifiche per determinati tipi di software.

2.3 Esempi di tassonomie

2.3.1 Tassonomia di Boris Beizer

Questa è una delle prime tassonomia dei bug, definita da Boris Beizer nel 1990. È molto dettagliata e tratta dei problemi del software a partire dalle specifiche di esso fino alla loro implementazione in codice e alla fase di testing.

Questa tassonomia identifica ogni categoria di bug con un numero a quattro cifre, dove ogni cifra rappresenta una categoria sempre più specifica. Sono utilizzate le x per indicare una macro-categoria che poi verrà suddivisa in categorie più specifiche. Ad esempio la categoria *4xxx* indica bug legati all'utilizzo o alla definizione delle strutture dati, e più nello specifico *411x* indica la sbagliata definizione del tipo di una variabile (ad esempio int al posto to float).

Dato che questa tassonomia include molte sotto-categorie (più di 150) di seguito verranno presentate solo le categorie principali:[8]

- **1xxx - Bug funzionali: requisiti e funzionalità**: vengono inclusi quei bug che dipendono da come i requisiti del software sono stati definiti o implementati. Appartengono a questa categoria i bug che si verificano quando le specifiche sono

sbagliate, illogiche, impossibili da implementare per motivi tecnici, tempistiche o economici.

- **2xxx - Funzionalità implementate**: bug che si verificano quando le specifiche sono corrette ma l'implementazione è errata. Vengono inclusi in particolare i bug derivanti da una sbagliata implementazione delle specifiche del software.

- **3xxx - Bug strutturali**: questa categoria include i bug correlati alla struttura del codice, ad esempio bug derivanti da un loop errati, cattiva gestione delle eccezioni, in generale quando il flusso di controllo del programma non è corretto.

- **4xxx - Strutture di dati**: qui sono inclusi tutti i bug riferiti all'utilizzo e definizione di strutture di dati. Ad esempio bug derivanti da variabili definite con tipo errato, array con lunghezza sbagliata, valori predefiniti incorretti, risorse non allocate o deallocate correttamente. Un esempio comune è l'integer overflow che si verifica quando il valore di una intero supera il massimo valore che può essere rappresentato dal tipo specifico.

- **5xxx - Implementazione**: bug derivanti da errori nell'implementazione del software; vengono inclusi i bug derivanti da errori di tipografici, violazione degli standard e convenzioni, errori nella documentazione.

- **6xxx - Integrazione**: bug collegati all'integrazione delle diverse componenti del software o l'integrazione con l'hardware. Ad esempio bug derivanti da errori di comunicazione tra componenti o errori di sincronizzazione.

- **7xxx - Architettura del sistema e software**: questa categoria include i bug non attribuibili ad un componente specifico ma che riguardano l'interno sistema.

- **8xxx - Definizione o esecuzione dei test**: Questa categoria include i bug presente nei test relativi alla loro definizione, esecuzione o documentazione.

2.3.2 Tassonomia bug dei videogiochi

Questa tassonomia è stata creata per classificare i bug che possono verificarsi nei videogiochi ed aiutare le decisioni sia dei game designer che dei tester a trovare i bug. A differenza della tassonomia presentata precedentemente, questa tassonomia deve essere utilizzabile anche dai soggetti che non hanno accesso al codice sorgente del software; infatti questa tassonomia non approfondisce le cause del problema, ma si basa su ciò che è direttamente osservabile da parte dell'utente. [9]

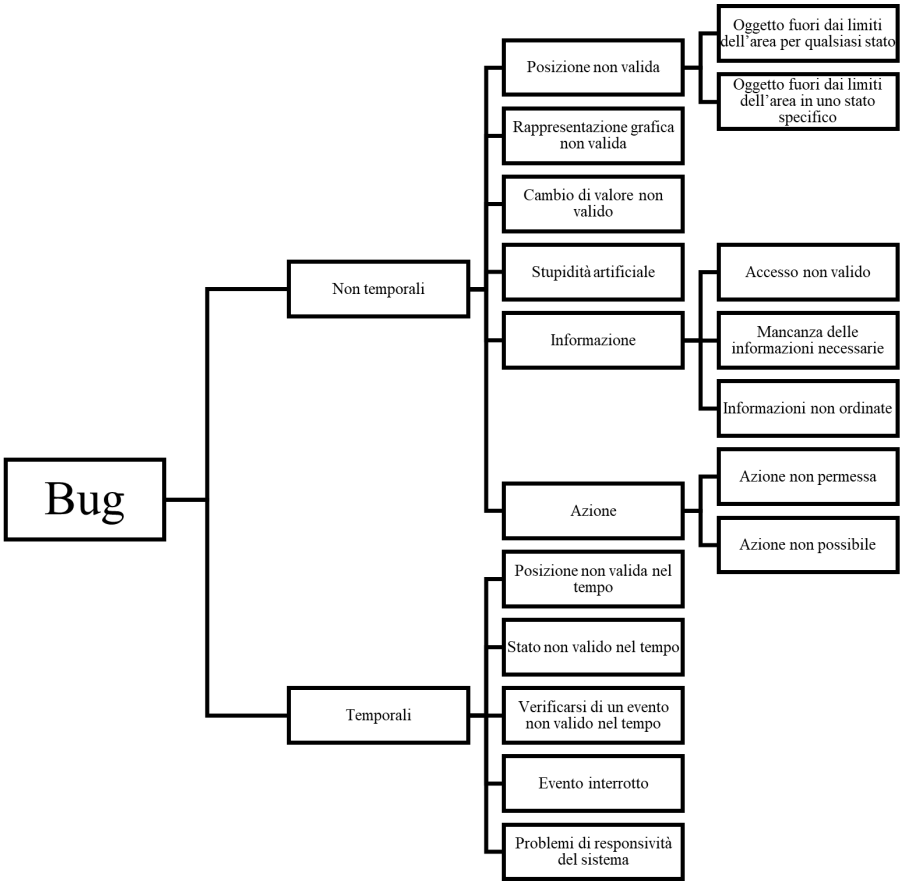

Figura 2.1: Tassonomia bug videogiochi

I bug vengono divisi in due categorie principali, temporali e non-temporali: i primi indicano i bug che per essere identificati richiedono una conoscenza dello stato precedente, mentre quelli non-temporali possono essere individuati in qualsiasi momento. Ad esempio se un oggetto si trova fuori dalla mappa è un bug non-temporale, mentre se un oggetto non si muove come dovrebbe è un bug temporale.

Partendo dai bug non-temporali si hanno le seguenti tipologie:

- **Posizione non valida**: un oggetto si trova fuori dall'ambiente di gioco.

- **Rappresentazione grafica non valida**: bug di questa tipologia avvengono quando un aspetto del ambiente non è renderizzato correttamente. Ad esempio viene mostrata l'animazione di nuoto anche se il personaggio è a terra.

- **Cambio di valore non valido**: questa è una categoria generale che si verifica quando un evento non ha l'effetto previsto. Un esempio è quando si ottiene un punto, ma il punteggio al posto di aumentare diminuisce.

- **Stupidità artificiale**: vengono inclusi tutti quei bug correlati agli NPC[1] che non si comportano come dovrebbero. Ad esempio un NPC impedisce al giocatore di proseguire bloccando il percorso.

- **Informazioni**: in questa categoria vengono incluse molteplici situazioni, quando il giocatore ottiene più informazioni di quelle che dovrebbe avere (ad esempio vedere attraverso i muri), quando il giocatore non ottiene le informazioni che dovrebbe avere ed in generale bug che includono la telecamera posizionata male, audio non presente o testo non visibile.

- **Azioni**: questa categoria include bug che permettono al giocatore di compiere delle azioni quando non potrebbe , oppure gli viene impedito di compiere azioni che dovrebbe poter fare.

Per quanto riguarda i bug temporali invece si hanno le seguenti tipologie:

- **Posizione non valida nel tempo**: in questa categoria si trovano bug relativi ai movimenti, come rapide accelerazioni, il giocatore sospeso in aria oppure bloccato in una posizione.

- **Stato non valido nel tempo**: per stato si intendono le caratteristiche di un oggetto/personaggio che sono visibili all'utente.

- **Verificarsi di un evento non valido nel tempo**: qui vengono classificati tutti quegli eventi che avvengono troppo frequentemente o troppo poco spesso.

- **Evento interrotto**: i bug in questa categoria si riferiscono ad eventi che erano in esecuzione, ma che vengono interrotti inaspettatamente.

- **Problemi di responsività del sistema**: questa categoria è la più vicina a come il gioco si interfaccia all'hardware, e si riferisce a quando il software non viene eseguito con le prestazione ottimali.

[1] Non-Playable-Character ovvero personaggi non controllati dal giocatore

2.3.3 Tassonomia dei bug dei sistemi di deep learning

I sistemi di deep learning sono modelli progettati per imparare da grandi quantità di dati e riconoscere pattern complessi. Questi sistemi utilizzano reti neurali artificiali con molti (deep) strati (layer) per elaborare informazioni, e prendere decisioni.

Con la diffusione dei sistemi di deep learning (DL) in moltissimi ambiti anche delicati come la medicina o guida autonoma, capire i problemi che possono verificarsi in questo tipo di sistemi è essenziale.

È da sottolineare che in questo tipo di sistemi i problemi possono verificarsi sia da come viene programmato il modello a livello di codice, sia dai dati che vengono forniti durante il training al modello; in questa tassonomia vengono analizzati entrambi gli scenari.[10]

La tassonomia è divisa in cinque categorie principali:

- **Modello**: questa categoria include bug relativi alla struttura e configurazione del modello. Problemi in questa categoria spesso coinvolgono la scelta sbagliata del tipo di modello (ad esempio utilizzare una CNN[2] al posto di una RNN[3]), inizializzazione errata dei parametri e struttura non corretta del modello (come il numero sbagliato dei layer).

- **Uso della GPU**[4]: i bug relativi a questa categoria includono ad esempio una scorretta gestione delle memoria della GPU, uso inefficiente del parallelismo. Questi problemi possono causare conseguenze come performance ridotte o costi computazionali maggiori.

- **API**[5]: questa categoria si riferisce all'uso errato delle funzioni fornite dai framework di DL come ad esempio TensorFlow, Keras e PyTorch. In questa categoria sono inclusi bug derivanti da utilizzo di funzioni deprecate, dipendenze non aggiornate o comunque un utilizzo non corretto delle API fornite.

- **Addestramento**: questa categoria affronta i problemi che si verificano durante la fase di addestramento, tra cui: scarsa qualità dei dati, errori durante il pretraining e iperparametri non ottimali.

- **Tensori ed input**: questa categoria include problemi derivanti dalla forma, tipo e formato dei dati, in particolare quando un tensore (una struttura matematica multidimensionale che generalizza scalari, vettori e matrici) ha la forma sbagliata

[2]Convolution Neural Network, un tipo di modello utilizzato per processare dati multidimensionali come le immagini

[3]Recursive Neural Network, un tipo di modello utilizzato per processare dati sequenziali come un testo

[4]Graphics Processing Unit

[5]Application Programming Interface

oppure quando un formato incompatibile viene usato come input in un layer. Alcuni esempi includono utilizzare un tensore 5x5 al posto di 5x10, oppure passare un decimale al posto di un intero.

2.3.4 Tassonomia vulnerabilità

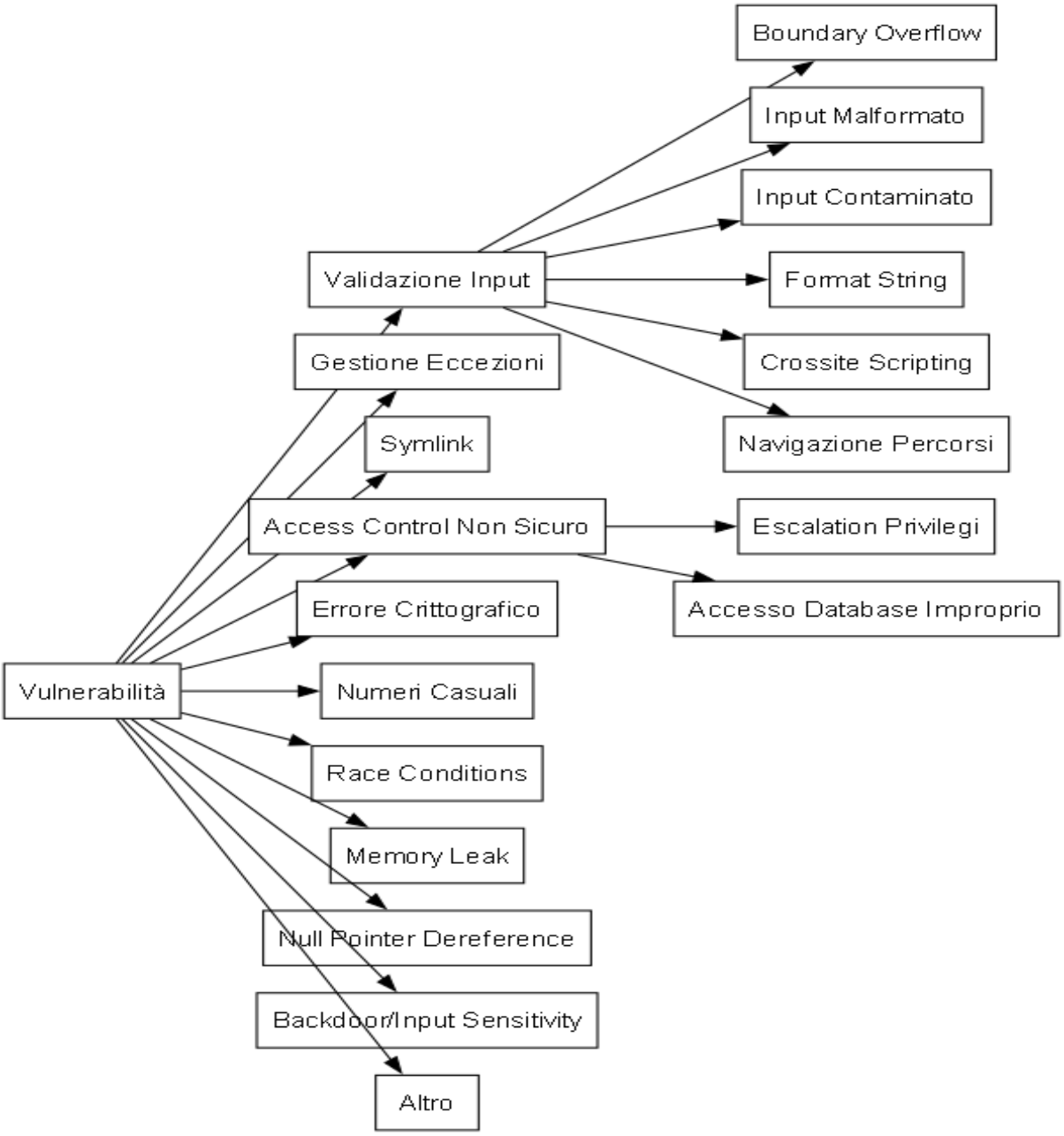

Figura 2.2: Tassonomia vulnerabilità

Questa tassonomia scritta dal NIST[6] classifica alcune vulnerabilità comuni sia nelle applicazioni server sia nei client. Comprende diverse categorie di vulnerabilità ed alcune sotto categorie.[11]

Alcune categorie nel dettaglio:

- **Errore validazione input**: quando gli input che vengono inviati ad un' applicazione non vengono verificati, questo può portare a diversi tipi di vulnerabilità come ad esempio *buffer overflow*, *sql o commandline injection* e *cross-site scripting*.

- **Access Control non sicuro**: in generale quando l'accesso all'applicazione è difettoso, e causa vulnerabilità come *privilege escalation* oppure i permessi degli utenti non sono impostati correttamente e permettono agli utenti di eseguire codice sul server.

- **Overflow**: quando l'input supera il limite di memoria allocata per esso, ad esempio l'applicazione utilizza più memoria di quella disponibile, oppure una variabile raggiunge il suo valore massimo e poi ritorna al suo valore minimo.

- **Race condition**: quando due o più processi o thread accedono e manipolano la stessa risorsa contemporaneamente le conseguenze possono essere imprevedibili, causando anche a potenziali vulnerabilità/

- **Altro**: si stima che il circa 21% delle vulnerabilità segnalate non siano classificate.

2.3.5 Tassonomia per la classificazione automatica dei bug

In questa sezione viene approfondita una classificazione presentata nello studio "Not All Bugs are the Same" [12] dove vengono analizzati diversi bug riportati in vari software open source e poi viene sviluppato un modello di machine learning per la classificazione automatica dei bug.

Le categorie di questa tassonomia sono:

- **Anomalie del programma**: bug che derivano da comportamenti imprevisti nella logica del programma o nell'implementazione degli algoritmi.

- **Problemi relativi alla GUI**[7]: bug associati all'interfaccia grafica dell'utente, come problemi di layout o pulsanti non funzionanti.

- **Problemi di configurazione**: errori nella configurazione del software.

[6] National Institute of Standards And Technology
[7] Graphical User Interface

- **Bug di sicurezza**: bug che rappresentano rischi per la sicurezza, potenzialmente permettendo accessi non autorizzati o compromettendo l'integrità dei dati.
- **Problemi di prestazioni**: problemi che degradano le prestazioni del software, come tempi di risposta lenti o alto consumo di risorse.
- **Problemi relativi al database**: bug relativi alla connessione tra il database a l'applicazione.
- **Problemi relativi alla rete**: bug causati da problemi di connessione o protocolli di comunicazione non utilizzati correttamente.
- **Problemi relativi al codice di test**: bug presenti nel codice di test.
- **Problemi relativi al codice deprecato**: bug associati all'utilizzo di API rimosse o deprecate e che quindi potenzialmente non funzionano più come dovrebbero.

Una classificazione adeguata dei bug facilita anche l'automazione del processo di categorizzazione degli stessi, anche tramite l'uso di tecniche di IA e machine learning. Gli algoritmi di machine learning vengono addestrati utilizzando il contenuto testuale dei report dei bug per prevedere la categoria di ciascun bug.

Gli approcci principali per categorizzare i bug sono:

- **Text mining**: analizzare le descrizioni dei report, estraendo le caratteristiche rilevanti per categorizzare il tipo di bug.
- **Apprendimento supervisionato**: Modelli come Naive Bayes e Support Vector Machine (SVM) vengono addestrati su un dataset etichettato di report dei bug, dove le categorie corrette sono già note.
- **Clustering e apprendimento semi-supervisionato**: Queste tecniche aiutano a migliorare il modello di classificazione identificando esempi informativi e migliorando l'accuratezza del modello.
- **Term Frequency-Inverse Document Frequency (TF-IDF)**: È un meccanismo di ponderazione che determina l'importanza delle parole nei report dei bug, migliorando la capacità del modello di identificare caratteristiche rilevanti.

I risultati dello studio dimostrano che il modello di classificazione automatica può categorizzare accuratamente i tipi di bug, raggiungendo un punteggio F-Measure[8] del 64% e un AUC-ROC[9] del 74% complessivamente. [12]

[8] indicare dell'accuratezza di un test

[9] Area Under The Curve - Receiver Operating Characteristics, è una misure di performance per modelli di classificazione

Questo livello di accuratezza evidenzia il potenziale dell'IA nel supportare gli sviluppatori automatizzando il processo iniziale di triage dei bug, riducendo così il tempo e lo sforzo necessari per affrontare risolvere il problema.

Capitolo 3

Bug significativi

3.1 Log4Shell

Log4Shell è stata definita come una delle vulnerabilità più gravi degli ultimi anni, scoperta nel 2021, ricevendo un punteggio CVSS[1] di 10/10[13] per diversi motivi tra qui la facilità con cui poteva essere sfruttata, l'enorme diffusione e la gravità delle conseguenze.

Questa vulnerabilità era presente in una libreria open source, tra le più utilizzate, chiamata *Log4j*, che viene utilizzata come sistema alternativo di logging a quello standard di Java. [14]

Il logging è l'atto di registrare dei messaggi (log) degli eventi che si verificano durante l'esecuzione di un software, che si tratti di errori o di semplici messaggi informativi. I log sono poi solitamente salvati in un file per essere consultati in seguito.

Ad esempio un server potrebbe volere registrare quando un utente esegue il login:

```
log.info("L'utente " + username + " ha effettuato l'accesso");
```

Una feature importante di Log4j sono i cosiddetti *lookup*, che consentono inserimento di valori dinamici all'interno di un log. I *lookup* sono stringhe del tipo `${nome:chiave}` dove il nome è il tipo di *lookup* e la chiave è il valore specifico da inserire, ad esempio con `${java:version}` si può inserire dinamicamente la versione di Java nel messaggio:

```
// ${java:version} potrebbe ad esempio essere sostituito durante
↪  l'esecuzione con "17.0.1"
log.info("La versione di java è: ${java:version}");
```

Nonostante i *lookup* siano utili, un particolare tipo di *lookup* chiamato *JNDI* (Java Naming and Directory Interface) introduce diversi rischi. *JNDI* è un API di Java che

[1] Common Vulnerability Scoring System, indica la gravità di una vulnerabilità in una scala da 0 a 10.

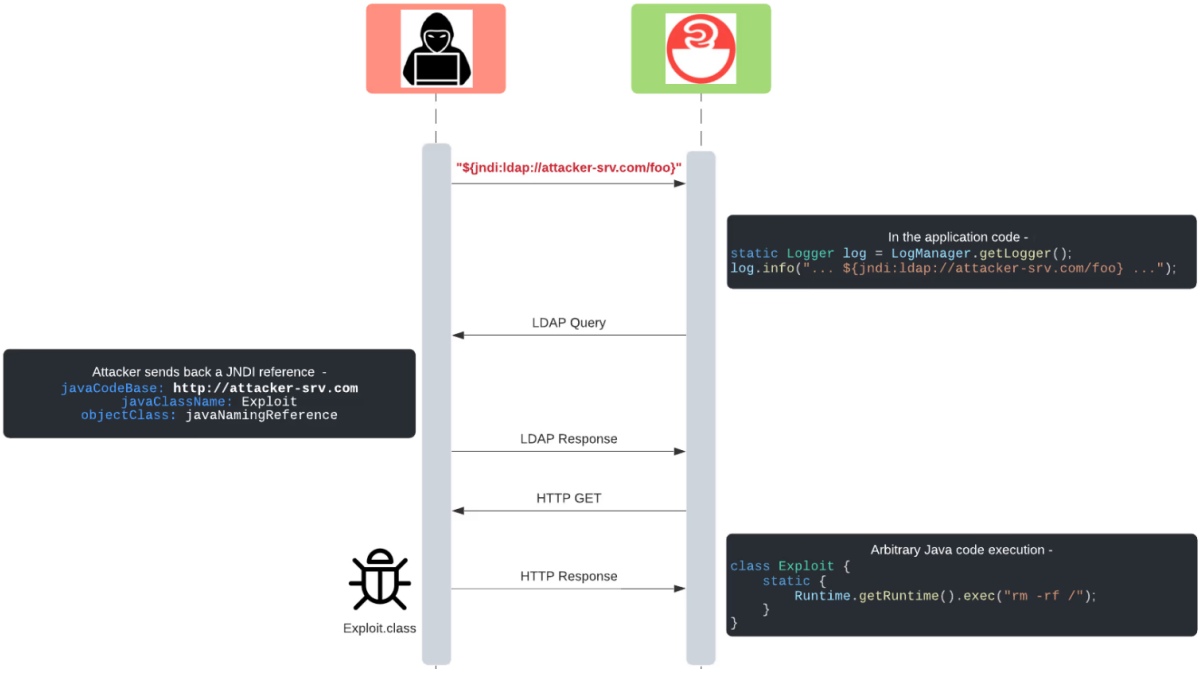

Figura 3.1: Exploit di Log4Shell
[15]

consente alle applicazioni di localizzare e recuperare risorse remote sotto forma di oggetti[2] Java.

Log4Shell consiste però nel ingannare l'applicazione a registrare un particolare messaggio affinché si connetta a un server controllato dall'attaccante, il quale invia del codice malevolo che viene eseguito dal server vulnerabile.

Per sfruttare *Log4Shell*, l'attaccante deve generalmente controllare l'input che *Log4j* registra; quest'ultimo può essere un messaggio di log, un'intestazione HTTP o qualsiasi altro dato fornito dall'utente. Inserendo un comando di *lookup JNDI* appositamente costruito, come `${jndi:ldap://attacker.com/a}`, l'attaccante indirizza *Log4j* a scaricare le risorse contenute nel server `attacker.com` e potenzialmente eseguire il codice che esse contengono. [16]

Riprendendo il primo esempio di logging, l'utente potrebbe inserire come username la stringa descritta sopra così che l'applicazione contatti il server dell'attaccante ed esegua il codice che esso fornisce. È quindi possibile classificare questo bug utilizzando la tassonomia del capitolo 2.3.4 come vulnerabilità causata da un errata validazione dell'input ed in particolare un input contaminato.

Log4shell (e potenziali varianti) può essere sfruttata per 3 potenziali tipi di attacchi:

[2]Un oggetto Java è un'istanza di una classe che incapsula dati e comportamenti correlati ad un'entità definita

1. **Remote Code Execution (RCE)**: eseguire codice arbitrario su un server vulnerabile.

2. **Information Disclosure**: ottenere informazioni, anche sensibili, dal server.

3. **Denial of Service (DoS)**: rendere il server inutilizzabile.

Questa vulnerabilità ha una superficie di attacco molto ampia perché consente di sfruttare qualsiasi sistema che registri i dati controllati dall'aggressore, rendendo così anche le applicazioni interne una superficie di attacco. Ciò significa che anche i sistemi apparentemente ben protetti o isolati da Internet possono diventare vulnerabili se elaborano e registrano una stringa dannosa creata per sfruttare *Log4Shell*. [14]

La gravità di questa vulnerabilità ha portato anche diverse istituzioni governative a dare avvertimenti sulla vulnerabilità e fornire possibili soluzioni. Gli enti che hanno emesso tali avvertimenti includono il CSIRT[3] Italia[17], l'NCSC[4] inglese e il CISA[5] americano [18]. L'FTC ha anche divulgato un avvertimento alle aziende che non avessero risolto la vulnerabilità tempestivamente, minacciando di intraprendere azioni legali contro chi non avesse risolto il problema.[19]

Log4shell è stato un evento che ha fatto riflettere su come, nonostante il software open-source formi le fondamenta di moltissimi software utilizzati oggi, molti di questi progetti siano mantenuti da poche persone e che spesso non ricevano il supporto necessario. Questo sembra paradossale se si pensa che aziende anche di dimensioni enormi da X (ex Twitter) ad Apple utilizzano software come *Log4Shell* senza contribuire al progetto. Infatti *Log4Shell* è stato creato da volontari e tutt'oggi viene mantenuto principalmente da volontari non pagati. [20]

La vulnerabilità di *Log4j* ha dimostrato come anche un progetto ampiamente utilizzato da migliaia di aziende possa essere vulnerabile a gravi problemi di sicurezza se non riceve adeguate risorse e attenzione.

Le organizzazioni no-profit, come la Apache Software Foundation e la Linux Foundation, già forniscono supporto, ma necessitano di ulteriori risorse per espandere le loro attività e garantire che i progetti critici siano adeguatamente finanziati e sostenuti. Alcuni passi in avanti sono stati fatti da grandi aziende come Google che nel 2021 ha stanziato 10 miliardi di dollari per incrementare la sicurezza dei software open-source[21], ma non sembra ancora sufficiente per risolvere il problema.

Log4Shell ha messo in evidenza non solo le vulnerabilità del software open-source, ma anche le debolezze della catena di fornitura del software (software supply chain). La catena

[3]Computer Security Incident Response Team
[4]National Cyber Security Center
[5]Cybersecurity and Infrastructure Security Agency

di fornitura del software comprende tutti i componenti, strumenti e processi utilizzati per sviluppare e distribuire un software. Ogni componente aggiunto alla catena introduce potenziali rischi di sicurezza, come dimostrato dall'incidente di *Log4shell*.

Nel contesto di *Log4j*, la vulnerabilità ha mostrato quanto sia critica la gestione dei componenti software di terze parti. Molti sviluppatori e aziende utilizzano librerie e dipendenze open-source come *Log4j*, senza una completa visibilità o comprensione delle potenziali vulnerabilità che possono introdurre. Questo ha portato a una diffusione massiccia del problema, con migliaia di applicazioni e servizi esposti a rischi di sicurezza.

Un concetto importante per mitigare questi rischi è quello dello SBOM (Software Bill of Materials). Lo SBOM è un elenco dettagliato di tutti i componenti inclusi in un software che consente agli sviluppatori di avere una visibilità completa sulle componenti del loro software, identificando rapidamente le dipendenze e valutando i rischi associati.

L'adozione di SBOM porta diversi vantaggi nel contesto della sicurezza della catena di fornitura del software. In primo luogo, facilita la gestione delle vulnerabilità. Con uno SBOM dettagliato, le aziende possono immediatamente identificare se una particolare vulnerabilità, come quella di *Log4shell*, impatta il loro software. Questo accelera il processo di patching e riduce il tempo di esposizione ai rischi.

Inoltre, lo SBOM migliora la trasparenza e la conformità. Molti settori regolamentati richiedono che le aziende dimostrino di avere controlli di sicurezza robusti. Avere uno SBOM ben definito può aiutare le aziende a soddisfare questi requisiti di conformità e a dimostrare la loro attenzione alla sicurezza. [22]

3.2 Y2K e Y38K

Il Millennium bug, o Y2K (year 2000) è stato un bug avvenuto quando si è passati dal 31 dicembre 1999 al 1 gennaio 2000 nei sistemi che gestivano le date utilizzando solo due cifre (in particolare sistemi degli anni '60 e '70) per risparmiare memoria. Con solo due cifre però era impossibile distinguere il 1900 dal 2000 e, con l'avvenire del nuovo secolo, i sistemi non aggiornati potevano potenzialmente andare in crash oppure tornare a 100 anni nel passato. [23]

Fortunatamente, grazie agli sforzi di prevenzione, i danni provocati da questo bug furono molto ridotti, ma comunque negli anni predenti al 2000 diverse controversie legali furono sollevate [24], in particolare riguardo a:

- **Responsabilità contrattuale**: Le aziende si sono trovate di fronte a potenziali violazioni contrattuali dovute alla mancata compliance riguardo al problema Y2K. I fornitori di software e hardware che non si sono assicurati che i loro prodotti

fossero conformi al problema Y2K sarebbero potuti essere ritenuti responsabili di violazione della garanzia. Ad esempio, casi come Atlaz International Ltd. contro Software Business Technologies Inc. hanno comportato accuse di violazione della garanzia perché il software non era in grado di riconoscere le date a partire dall'anno 2000.

- **Responsabilità extracontrattuale**: Le aziende potrebbero essere ritenute responsabili di negligenza se sapevano o avrebbero dovuto sapere del Y2K e non hanno preso provvedimenti adeguati. Nella causa Faegenburg contro Intuit, Inc. è stato affermato che l'industria informatica era a conoscenza del problema dell'anno 2000 fin dai primi anni Settanta, ma non ha fatto nulla a causa dei costi aggiuntivi che avrebbe comportato. Ciò ha introdotto il concetto di conoscenza collettiva del settore e la possibilità di imputare la responsabilità in base allo stato del settore.

- **Assicurazione**: Sono state sollevate anche questioni relative all'estensione della copertura fornita da varie polizze assicurative. Le aziende hanno cercato di ottenere una copertura nell'ambito di diverse polizze assicurative come quelle contro l'interruzione dell'attività e assicurazione di responsabilità civile generale. Le compagnie di assicurazione, a loro volta, erano preoccupate riguardo alle potenziali numerose richieste di risarcimento derivanti dalle interruzioni legate allo Y2K.

Una situazione simile potrebbe verificarsi nel 2038, il cosiddetto Y38K, quando i sistemi che utilizzano il formato Unix time, ovvero il numero di secondi trascorsi dal 1 gennaio 1970, e usano un intero con segno a 32 bit per tenere traccia di tale valore raggiungeranno il valore massimo rappresentabile da tale tipo, ovvero $2^{31} - 1$, che corrisponde alla data `03:14:07 UTC`[6] `del 19 Gennaio 2038`; dopo quel momento molti sistemi di questo tipo potrebbero andare in crash oppure non gestire correttamente la data, ad esempio trasformando la data nel `20:45:52 UTC del 13 Dicembre 1901` [25]

Il motivo di tale comportamento è dovuto al fatto che gli interi con segno a 32bit utilizzano il bit più significativo per rappresentare il segno, quindi quando questo valore viene incrementato di uno oltre il limite massimo, il bit più significativo viene impostato a 1 e l'intero viene interpretato come negativo, assumendo il valore minimo rappresentabile (questo è dovuto al fatto che i numeri interno con segno vengono rappresentati con il metodo del complemento a due). Si può notare la transazione dalla figura 3.2 dove tutti i bit tranne quello del segno sono a 1, a quella della figura 3.3 dove il bit del segno è a 1 e tutti gli altri sono a 0, e quindi quando alla data contenuta nella prima figura verrà aggiunto un secondo si passerà dal valore massimo rappresentabile a quello minimo.

[6] Universal Time Control

```
Binary  : 01111111 11111111 11111111 11111111
Decimal : 2147483647
Date    : 2038-01-19 03:14:07 (UTC)
Date    : 2038-01-19 03:14:07 (UTC)
```

Figura 3.2: Prima dell'overflow

```
Binary  : 10000000 00000000 00000000 00000000
Decimal : -2147483648
Date    : 1901-12-13 20:45:52 (UTC)
Date    : 2038-01-19 03:14:08 (UTC)
```

Figura 3.3: Dopo l'overflow

I bug Y2K che Y38K possono entrambi essere classificati con la tassonomia illustrata nel capitolo 2.3.1 come bug appartamenti alla categoria dei bug relativi alle strutture dati, in quanto nel primo caso l'anno veniva rappresentato con solo due cifre e quindi insufficiente per rappresentare il passaggio al nuovo secolo, mentre nel caso di Y38K si può parlare nello specifico di un integer overflow, in quanto si cerca di rappresentare un numero maggiore di del massimo che può essere rappresentato da quel tipo di dato.

3.3 Knight Capital Group

Knight Capital Group (KCG) era una grande azienda americana di servizi finanziari. Nel 2012, era diventata il più grande trader di azioni negli Stati Uniti, con una quota di mercato di circa il 17% sia sulla Borsa di New York (NYSE) che sul Nasdaq Stock Market. Il gruppo di trading elettronico dell'azienda gestiva un impressionante volume di trading giornaliero di oltre 3,3 miliardi di operazioni, per un valore superiore a 21 miliardi di dollari. [26]

Il primo agosto 2012, un bug nel loro software ha causato a Knight Capital Group un incidente che nel giro di un ora ha quasi distrutto l'azienda.

L'evento è stato innescato da un malfunzionamento del nuovo software di trading, attivato all'apertura della NYSE quel giorno. Questo errore del software ha portato i sistemi di KCG ad erroneamente acquistare 150 diverse azioni per un costo totale di circa 7 miliardi di dollari nella prima ora di trading.

La causa principale dell'incidente è stata un guasto del software nel sistema di trading di Knight, SMARS (Smart Market Access Routing System). Questo guasto è stato dovuto a due principali motivi:

1. **Problema del Codice Power Peg**: Il nuovo codice RLP (Retail Liquidity Program) è stato distribuito sostituendo un vecchio codice non utilizzato per un algoritmo chiamato "Power Peg", che Knight non usava dal 2003. Power Peg era un algoritmo per fare dei test, progettato per acquistare azioni ad un prezzo elevato e rivendere ad un prezzo basso per provocare uno spostamento del valore dalle azioni. Questo serviva a testare gli altri algoritmi in un ambiente controllato, ed infatti

non era destinato al trading live. Tuttavia, questo vecchio codice non è stato correttamente rimosso ed è rimasto eseguibile. Questo problema, facendo riferimento alla tassonomia del capitolo 2.3.1, può essere classificato come un bug strutturale il quanto il codice che non più utilizzato sarebbe dovuto essere rimosso e non essere più eseguibile.

2. **Distribuzione inconsistente del nuovo codice**: KKG disponeva di 8 server sul quale doveva essere installato il nuovo software RLP, ma non era previsto un sistema di distribuzione automatica e il tecnico che si occupò dell'installazione del nuovo software lasciò un server non aggiornato creando uno stato inconsistente tra i server. Utilizzando la tassonomia al capitolo 2.3.1 si può classificare questo problema come un bug di architettura del sistema in quanto esso non era stato progettato per gestire in modo corretto la distribuzione del software.

I tentativi di Knight di annullare le operazioni errate sono stati in gran parte negati dalla SEC[7], che ha solo annullato sei transazioni che superavano la soglia di annullamento del 30% di aumento del prezzo. Di conseguenza, Knight è stata costretta a vendere le azioni acquisite, subendo una perdita massiccia.

Goldman Sachs è intervenuta, acquistando le posizioni indesiderate di Knight con una perdita di 440 milioni di dollari, prevenendo un fallimento immediato ma lasciando l'azienda finanziariamente debilitata. Nonostante l'intervento di Goldman Sachs però KKG rimase in uno stato precario.

Dopo un investigazione della SEC, KKG è stata multata di 12 milioni di dollari e obbligata a sostenere un analisi indipendente per verificare la compliance con le regole del mercato. La sanzione è stata applicata per diversi motivi tra cui: non aver implementato di un sistema di gestione rischi efficace, controlli e procedure per l'installazione del loro software, procedure per aiutare i dipendenti a gestire le situazioni di emergenza e infine non aver verificato la compliance le *market access rules*.

Alla fine, Knight Capital Group è stata acquisita da Virtu LLC nel luglio 2017 per 1,4 miliardi di dollari. [26]

Questo evento aiuta a capire la necessità di implementare diversi sistemi di controllo e di testare il software prima di metterlo in produzione, in particolare in settori dove un errore può avere conseguenze gravi.

Alcune tra le procedure che avrebbero potuto evitare l'evento sono:

- **Utilizzare un sistema di versionamento del codice**: il codice non utilizzato dovrebbe essere rimosso e un sistema di versionamento (VCS) utilizzato per tenere

[7] Security and Exchange Commission

traccia delle modifiche. Un sistema di versionamento è uno strumento utilizzato per registrare le modifiche del codice nel tempo e gestire le diverse versioni del software, il più famoso di questi strumenti è *Git*.

- **Controllo del software**: esaminare accuratamente il software tramite ad esempio tramite peer review. La peer review è una pratica di sviluppo del software in cui il codice scritto da un programmatore viene esaminato da altri programmatori per identificare errori o applicare miglioramenti.

- **Test del software**: implementare test automatizzati per verificare il corretto funzionamento del software, ed in generale prevedere un sistema di CI/CD (Continuous Integration/Continuous Deployment). La continuous integration (letteralmente integrazione continua) è una pratica che consiste nel integrare spesso il nuovo codice nel VCS e testare automaticamente ogni modifica effettuata, mentre il continuos deployment (letteralmente distribuzione continua) è una pratica che consiste nel rilasciare in automatico gli aggiornamenti del software dopo che quest'ultimo ha passato tutti i test automatici.

- **Distribuzione del software automatica**: implementare un sistema di distribuzione automatica del software per evitare di dover manualmente installare il software su molti server e prevenire errori umani.

3.4 Therac-25

Il Therac-25 era un dispositivo medico progettato dall'AECL[8] per trattare il cancro mediante la somministrazione di radiazioni ai tumori che è entrata in servizio nel 1982. Era una macchina avanzata che poteva utilizzare sia elettroni che raggi X per colpire le cellule tumorali, fornendo trattamenti efficaci cercando di minimizzare i danni ai tessuti circostanti. [27]

Tra il 1985 e il 1987 però ci sono stati sei gravi incidenti che hanno coinvolto il Therac-25. Questi incidenti sono avvenuti a causa di gravi difetti nel software e nel design della macchina. I problemi hanno portato i pazienti a ricevere dosi massicce di radiazioni, molto oltre ciò che era previsto per i loro trattamenti. Gli incidenti principali furono:

- **Kennestone Regional Oncology Center (Marietta, Georgia), 1985:** Il 3 giugno 1985, una donna di 61 anni in trattamento radioterapico per un tumore al seno ha avvertito un bruciore durante la seduta. Inizialmente non vi erano segni

[8] Atomic Energy of Canada Limited

visibili, ma in seguito ha sviluppato un significativo rossore, gonfiore e dolore nella zona trattata. Il fisico dell'ospedale ha stimato che avesse ricevuto una dose di radiazioni tra i 15.000 e i 20.000 rad, ben oltre la dose terapeutica tipica di 200 rad. Questa overdose ha portato alla perdita di funzionalità delle spalle, la rimozione del seno oltre un significativo dolore. Nonostante queste lesioni, la causa non è stata immediatamente attribuita al Therac-25 e non è chiaro se AECL fosse a conoscenza dell'evento dopo che è accaduto se non quando nel 1986 il paziente intraprese un azione legale contro AECL che venne risolta fuori dal tribunale. Dopo questo incidente AECL ha continuato a ritenere che fosse impossibile che Therac-25 potesse erogare una dose di radiazioni così elevate. I dettagli di questi caso rimangono ancora poco chiari in quanto non avvenne mai una vera e propria indagine.

- **Ontario Cancer Foundation (Hamilton, Ontario), 1985:** Il 26 luglio 1985, durante il trattamento di una paziente per un carcinoma della cervice, la macchina subì diversi arresti inaspettati e secondo il display nessuna dose di radiazioni fu erogata, questo era un comportamento comune della macchina ma che fino ad allora non aveva mai causato danni ai pazienti. In questo caso Therac-25 fu riavviata 4 volte finche la donna ha avvertito un forte dolore, gonfiore e sensazioni di bruciore. AECL non è stata capace di replicare questo problema, ma sospettava di un problema in un microswitch[9] che determinava la posizione di alcuni componenti della macchina, questo problema fu risolto con un aggiornamento software, ma comunque AECL ha dichiarato che non era a conoscenza delle cause del problema e poteva solo supporre che questo fosse il reale motivo del malfunzionamento.

- **Yakima Valley Memorial Hospital (Yakima, Washington), 1985:** Nel dicembre 1985, una paziente trattata presso lo Yakima Valley Memorial Hospital ha sviluppato eritema sull'anca destra dopo un trattamento con il Therac-25. La causa non è stata immediatamente riconosciuta e la paziente ha continuato i trattamenti fino a gennaio 1986. In seguito si è sospettato che fosse stata sovraesposta alle radiazione, ma questo è stato confermato solo un anno dopo, a seguito di un altro incidente di sovraesposizione nello stesso ospedale, in quanto inizialmente dopo che l'ospedale aveva segnalato l'evento all'AECL gli era stato risposto che era da escludere un malfunzionamento del Therac-25.

- **East Texas Cancer Center (Tyler, Texas), marzo e aprile 1986:** Due pazienti morirono a causa di una overdose di radiazioni subita durante il trattamento di un tumore, a causa di un errore del software che si verifica quando la modalità

[9] un piccolo interruttore a scatto

veniva cambiata da X (raggi X) ad e (elettroni). Dopo l'avvio della macchina, la scritta "Malfunction 54" appariva sul display, e veniva indicato che una dose piccola di radiazioni era stata emessa ma in realtà la dose emessa era massiccia. Dopo questi eventi la macchina è stata temporaneamente disattivata per investigare sui malfunzionamenti. La causa di questo problema è stata poi identificata come una race condition: per attivare la macchina erano necessarie due fasi quella di inserimento dei dati e quella di posizionamento del magnete, quest'ultima richiedeva 8 secondi per esse completata. Quindi se l'operatore in quel lasso di tempo cambiava determinate impostazioni la macchina non le aggiornava correttamente potenzialmente erogando una dose errata di radiazioni.

- **Yakima Valley Memorial Hospital (Yakima, Washington), gennaio 1987:** Il 17 gennaio 1987, un paziente del Yakima Valley Memorial Hospital doveva sottoporsi ad un trattamento per un carcinoma. Il tecnico quindi ha proceduto ad impostare la macchina per il trattamento specifico, ma dopo che essa è stata attivata nonostante il raggio era visibile nessun informazione veniva visualizzata sulla console, nonostante ciò si è proseguito con il trattamento fino a che la donna non ha avvertito una sensazione di bruciore sul petto. Un indagine successiva determinò che invece di erogare gli 86rad prescritti, furono invece erogati tra gli 8000 e i 1000 rad. In questo caso AECL non ha dichiarato espressamente quale fosse il problema.

Questi eventi evidenziano i gravi difetti del software del Therac-25, i quali AECL ha provato a risolver implementato un aggiornamenti software, l'installazione di sistemi hardware di sicurezza come un nuovo sensore per gestire il posizionamento della macchina, e un sistema di interblocco che avrebbe impedito alla macchine di mettere a rischio i pazienti. [27] [28]

Alcuni riflessioni sulla base di questi incidenti:

- **Troppa fiducia nel software**: Affidarsi al software per funzioni di sicurezza critiche, senza implementare anche sistemi di sicurezza a livello hardware, può portare a gravi conseguenze.

- **Ingegneria del software**: Diverse principi base dell'ingegneria del software sono stati violati nel caso del Therac-25, tra cui: una documentazione adeguata, verifica della qualità del software, design semplice, messaggi di errore chiari e test del software approfonditi.

- **Regolamentazioni adeguata**: Una forte supervisione regolamentare è essenziale per garantire che i dispositivi medici siano sicuri per l'uso.

- **Prospettiva tecnica e legale**: Nelle azioni legali relative a malfunzionamenti, non sempre si viene a conoscenza di quale fosse lo specifico problema tecnico. La trasparenza può essere limitata dalla riservatezza mantenuta dalla casa produttrice. Dal punto di vista tecnico è fondamentale comprendere e classificare la causa del bug, per prevenire futuri incidenti, invece dal punto di vista legale invece l'obiettivo è determinare la responsabilità del danno. Bisogna quindi bilanciare la necessità di trasparenza con la protezione delle informazioni riservate del produttore.

Gli incidenti del Therac-25 evidenziano anche come focalizzarsi su singoli bug del software non è sufficiente per garantire la sicurezza del sistema. Il vero problema risiede nelle pratiche di ingegneria del software inefficaci e nella progettazione insicura del sistema nel suo complesso. Un software complesso può comportarsi in modo imprevedibile e l'affidamento del Therac-25 sul software per la sicurezza, senza adeguati sistemi hardware di protezione si è rivelato non efficace a prevenire vittime.

Utilizzando la tassonomia di Boriz-Beizer i vari bug accaduti nel Therac-25 possono essere classificati come:

- **Bug strutturali**: sono diversi i bug correlati alla struttura del software utilizzato in Therac-25, in particolare si può evidenza la race condition che ha causato il problema avvenuto nel Yakima Valley Memorial Hospital.

- **Integrazione**: molteplici sono i bug che hanno avuto origine da problemi di integrazione tra il software e l'hardware del sistema.

3.5 Boeing 737 MAX

Il Boeing 737 MAX è stato coinvolto nel 2018 e 2019 in due incidenti che hanno causato la morte di tutti i passeggeri a bordo di entrambi i voli.

Il Boeing 737 ha fatto il suo debutto nel 1967 come un aereo relativamente semplice e motori di dimensioni discrete. Nel corso del tempo, il modello si è evoluto, diventando più grande e complesso per soddisfare le esigenze del mercato e i progressi tecnologici. Il Boeing 737 MAX venne creato per aumentare l'efficienza del modello precedente, installando motori di dimensioni più grandi.

Per introdurre i nuovi motori, Boeing ha dovuto apportare diverse modifiche al design dell'areo, incluso lo spostamento dei motori in avanti e verso l'alto. Questa modifica, però, ha cambiato le proprietà aerodinamiche dell'aereo causando una notevole alzata del muso, in particolare durante il decollo e a basse velocità.

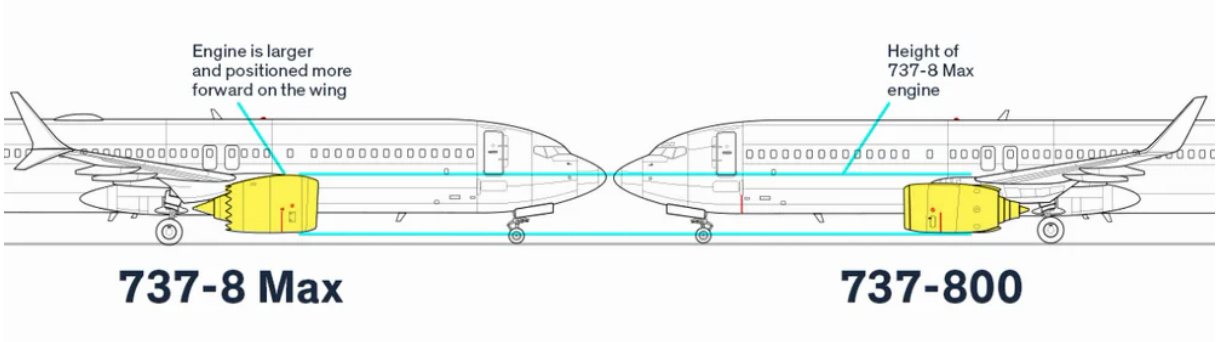

Figura 3.4: Boeing 737 MAX
[29]

Per contrastare la tendenza del muso a sollevarsi causata dal nuovo posizionamento dei motori, Boeing ha cercato di risolvere il problema senza cambiare la struttura dell'aereo, ma introducendo un software chiamato Maneuvering Characteristics Augmentation System (MCAS). Questo sistema è stato progettato per abbassare automaticamente il muso dell'aereo se rilevava che l'angolo di incidenza (AoA[10]) era troppo alto, evitando così uno stallo aerodinamico.

Tuttavia il sistema il sistema MCAS si rivelò difettoso causando due incidenti fatali: furono coinvolti il volo Lion Air 610 e Ethiopian Airlines 302, rispettivamente nel ottobre 2018 e marzo 2019. Entrambi gli incidenti hanno coinvolto aerei 737 MAX e hanno provocato la perdita di 346 vite in totale. Le indagini hanno rivelato che in entrambi i casi i sensori AoA hanno riportato dati errati attivando il sistema MCAS, portando quindi a inclinazioni incontrollabili verso il basso e, dato che il sistema era progettato per ignorare anche gli input dati dal pilota fu impossibile prevenire lo schianto.

Questo tipo di problema è quindi classificabile come un difetto di integrazione tra i sensori dell'aereo e il software MCAS, dato che quest'ultimo non era capace di distinguere i dati corretti da quelli errati.

Questi incidenti hanno rivelato difetti significativi non solo nel sistema MCAS, ma anche nell'approccio di Boeing alla certificazione degli aerei e alla formazione dei piloti. Boeing aveva commercializzato il 737 MAX come una nuova versione dei modelli precedenti del 737, evitando la necessità di ulteriore formazione per i piloti, in questo modo il costo per riqualificare i piloti sostenuto dalle compagnie aeree sarebbe stato notevolmente ridotto.

Inoltre la FAA[11] aveva sempre più delegato le responsabilità di certificazione della sicurezza ai produttori attraverso i Rappresentanti Ingegneristici Designati (DER). Questo

[10] Angle of Attack
[11] Federal Aviation Administration

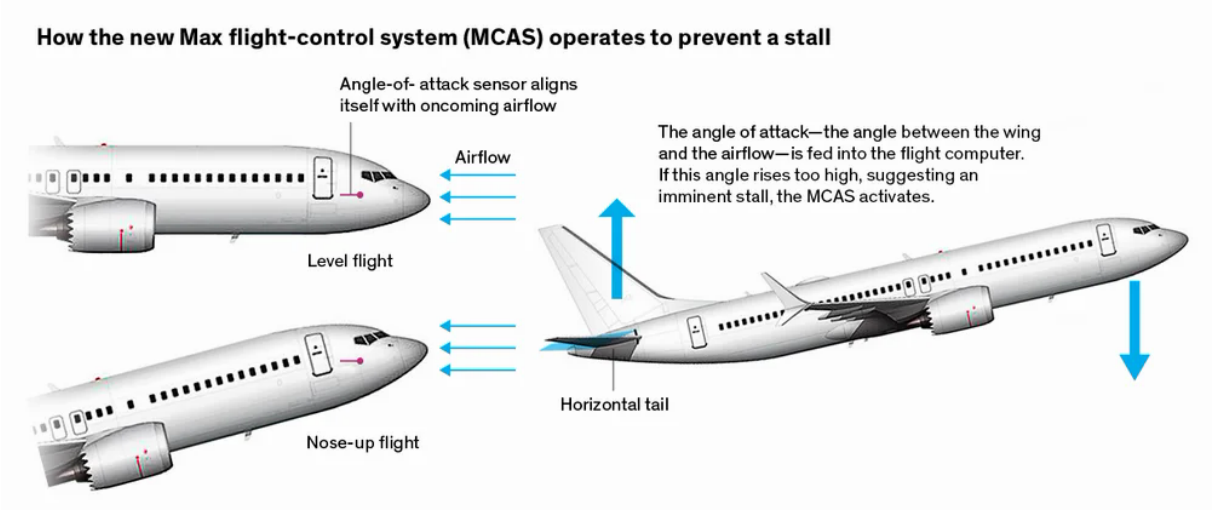

Figura 3.5: Sistema anti-stallo del Boeing 737 MAX
[29]

sistema si affidava ai produttori per autocertificare la sicurezza dei loro progetti, creando potenziali conflitti di interesse.

Nel caso del 737 MAX, la mancanza di controlli incrociati tra i diversi sensori e l'impossibilità del pilota di sovrascrivere le operazioni compiute della machine erano problemi significativi che però avrebbero dovuto essere rilevate durante il processo di certificazione, ma la pressione di voler introdurre sul mercato l'aereo più in fretta possibile probabilmente porto a minimizzare questi aspetti.

In risposta agli incidenti, Boeing ha iniziato a lavorare su aggiornamenti software per il sistema MCAS per incorporare controlli incrociati e ridurre la sua priorità, rendendo più facile per i piloti prendere il controllo. Inoltre, si è posta più attenzione sull'assicurarsi che tutti i sistemi e le procedure critiche per la sicurezza siano comunicati in modo trasparente e compresi dai piloti. [29]

In seguito agli incidenti diversi paesi, tra cui Stati Uniti e Cina, sospesero i voli del 737 MAX per almeno un anno e Boeing subì una perdita complessiva di $18.4 miliardi [30] e la cancellazione di 183 ordini di aerei.

Riguardo a questo caso, il 10 luglio 2024, Boeing si è dichiarata colpevole in un patteggiamento con il Dipartimento di Giustizia degli Stati Uniti riguardo ad una processo iniziato ad ottobre di questo anno. Boeing era stata accusata di non aver rispettato degli accordi stipulati nel 2021 che prevedevano che la società apportasse modifiche alla sicurezza degli aerei e si implementasse processi produttivi più trasparenti.

La società ha quindi accettato di dichiararsi colpevole di aver cospirato per truffare la FAA, ed è stata sanzionata con una multa di 478,2 dollari e dovrà anche investire almeno 455 milioni di dollari per rafforzare i programmi di sicurezza. [31]

Capitolo 4

Aspetti legali

4.1 Cyber Resilience Act

Il Cyber Resilience Act (CRA) dell'Unione Europea è un regolamento approvato dal Parlamento Europeo a marzo 2024, che dovrebbe entrare in vigore nella seconda metà del 2024. Esso mira a migliorare la sicurezza informatica dei prodotti digitali disponibili nel mercato interno.

Questa legislazione affronta le crescenti minacce informatiche che accompagnano la proliferazione dei dispositivi e servizi digitali, garantendo un maggiore livello di sicurezza per consumatori e imprese. Il regolamento si applica a tutti i software e prodotti connessi alla rete (ad eccezione di quelli per cui esiste già una normativa specifica, come ad esempio i dispositivi medici e dei software distribuiti come servizi e non come prodotti), i quali dovranno dotarsi del marchio CE per dimostrare di applicare i nuovi standard; questo sarà un requisito per distribuire il prodotto nel mercato unico.

Il CRA opera sia dal punto di vista tecnico che legale. Dal punto di vista tecnico, esso stabilisce degli standard di sicurezza informatica che tutti i prodotti digitali devono rispettare. Questo include la progettazione, lo sviluppo e l'implementazione di misure di sicurezza durante l'intero ciclo di vita del prodotto, dalla produzione alla manutenzione post-vendita. Inoltre, la normativa introduce un sistema di categorizzazione basato sul rischio, che classifica i prodotti in base al loro livello di rischio informatico e impone valutazioni di conformità più rigorose per i prodotti ad alto rischio. Alcuni obblighi tecnici che la normativa impone sono: proteggere l'integrità dei dati, raccogliere solo i dati necessari per il funzionamento del prodotto e fornire un software sicuro by design che deve ricevere aggiornamenti di sicurezza per tutto il ciclo di vita del prodotto. [32]

Dal punto di vista legale, il CRA armonizza i requisiti di sicurezza informatica in tutta l'UE, prevenendo che gli Stati membri impongano ulteriori regolamenti che potrebbero

frammentare il mercato unico. La normativa garantisce quindi che tutti gli attori economici nella catena di fornitura rispettino le stesse regole e standard di sicurezza. Questo include l'obbligo per i fornitori di software di implementare un processo completo di gestione delle vulnerabilità completo, inclusa la loro classificazione e documentazione. Il CRA inoltre facilita l'accesso al mercato unico per le imprese e protegge i consumatori da prodotti digitali insicuri, promuovendo al contempo la trasparenza e l'accesso alle informazioni sulla sicurezza dei prodotti.

Si possono riassumere gli obbiettivi principali del CRA in quattro punti:

- **Migliorare la sicurezza dei prodotti**: Assicurare che tutti i prodotti digitali, siano essi elettronica di consumo o apparecchiature industriali, rispettino rigorosi standard di sicurezza informatica durante tutto il ciclo di vita del prodotto.

- **Categorizzazione basata sul rischio**: Implementare un sistema di categorizzazione in cui i prodotti sono classificati in base al loro livello di rischio informatico. I prodotti ad alto rischio sono sottoposti a rigorose valutazioni di conformità da parte di enti indipendenti.

- **Sicurezza della catena di fornitura**: Enfatizzare la sicurezza dell'intera catena di fornitura, richiedendo ai produttori di garantire la sicurezza informatica sia dei prodotti direttamente connessi sia di quelli indirettamente connessi, come ad esempio dipendenze o librerie utilizzate da un software.

- **Armonizzazione dei regolamenti**: Standardizzare i requisiti di sicurezza informatica in tutta l'UE per facilitare un accesso al mercato più agevole e prevenire che gli stati membri impongano ulteriori regolamenti.

Il regolamento mira a proteggere da vulnerabilità legate ai prodotti digitali, mitigare le perdite finanziarie per le imprese dovute a violazioni della sicurezza informatica e garantire che l'intera catena del software sia sicura.

Il CRA richiede quindi ai fornitori di software di avere un processo di gestione delle vulnerabilità completo, inclusa la classificazione e la documentazione delle vulnerabilità del software. Una tassonomia dei bug del software ben progettata può risultare molto utile nell'aiutare le organizzazioni a soddisfare i requisiti del CRA in diversi modi:

- **Classificazione consistente e completa dei difetti**: una tassonomia fornisce un modo consistente per classificare i difetti del software in base al loro tipo, gravità e potenziale impatto sulla sicurezza. Questo consente agli sviluppatori la possibilità di identificare, tracciare e rimediare più efficacemente le vulnerabilità durante tutto il ciclo di vita del prodotto.

- **Prioritizzazione dei difetti critici per la sicurezza**: classificando i difetti in base al loro potenziale impatto sulla sicurezza, una tassonomia aiuta le organizzazioni a concentrare i loro sforzi sulle vulnerabilità più critiche che rappresentano il maggior rischio per gli utenti. Questo è in linea con l'obiettivo del CRA di ridurre le vulnerabilità diffuse nei prodotti digitali.

- **Automatizzazione della classificazione delle vulnerabilità**: integrando tecniche di machine learning nella tassonomia, è possibile classificare automaticamente i report di bug come problemi che possono influire sulla sicurezza del software o meno. Questo semplifica il processo di gestione delle vulnerabilità e aiuta le organizzazioni a identificare e affrontare efficientemente i difetti rilevanti per il CRA.

È comunque necessario ribadire che non è possibile definire una tassonomia univoca e standardizzata per tutti i tipi di software, ma è necessario che essa si adatti al software specifico dove viene applicata, in particolare riguardo alle vulnerabilità di sicurezza esse possono variare notevolmente a seconda del contesto di esecuzione del software.

Bibliografia

[1] GeeksforGeeks. *Software Testing – Bug vs Defect vs Error vs Fault vs Failure*. 2023. URL: www.geeksforgeeks.org/software-testing-bug-vs-defect-vs-error-vs-fault-vs-failure/.

[2] Oxford Learner's Dictionaries. *glitch*. 2024. URL: www.oxfordlearnersdictionaries.com/definition/english/glitch_1.

[3] National Museum of American History. *Log Book With Computer Bug*. 1947. URL: https://americanhistory.si.edu/collections/nmah_334663.

[4] Oxford English Dictionary. *bug, n.², sense II.5.a*. 2024. URL: https://doi.org/10.1093/OED/8132661290.

[5] Fred R. Shapiro. «Etymology of the Computer Bug: History and Folklore». In: *American Speech* 62.4 (1987), pp. 376–378. ISSN: 00031283, 15272133. URL: http://www.jstor.org/stable/455415.

[6] Herb Krasner. «The Cost of Poor Software Quality in the US: A 2020 Report». In: (2020). URL: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf.

[7] Margarita Simonova. *Costly Code: The Price Of Software Errors*. 2023. URL: https://www.forbes.com/sites/forbestechcouncil/2023/12/26/costly-code-the-price-of-software-errors/.

[8] Boris Beizer as amended by Otto Vinter. *Bug taxonomy and software statistics*. URL: http://ottovinter.dk/bugtaxst.doc.

[9] Chris Lewis, Jim Whitehead e Noah Wardrip-Fruin. «What Went Wrong: A Taxonomy of Video Game Bugs». In: (giu. 2010). DOI: 10.1145/1822348.1822363.

[10] Nargiz Humbatova et al. *Taxonomy of Real Faults in Deep Learning Systems*. 2019. arXiv: 1910.11015.

[11] NIST. *A Taxonomy of Software Flaws*. 2021. URL: https://www.nist.gov/itl/ssd/software-quality-group/taxonomy-software-flaws.

[12] Gemma Catolino et al. «Not all bugs are the same: Understanding, characterizing, and classifying bug types». In: *Journal of Systems and Software* 152 (2019), pp. 165–181. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.03.002. URL: https://www.sciencedirect.com/science/article/pii/S0164121219300536.

[13] *CVE-2021-44228*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-44228.

[14] Douglas Everson, Long Cheng e Zhenkai Zhang. «Log4shell: Redefining the web attack surface». In: *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*. 2022, pp. 1–8.

[15] Shachar Menashe, Or Peles e Ori Hollander. *Log4j Log4Shell 0-Day Vulnerability: All You Need To Know*. URL: https://jfrog.com/blog/log4shell-0-day-vulnerability-all-you-need-to-know/.

[16] Hans-Martin Münch. *Vulnerability notes: Log4Shell*. URL: https://mogwailabs.de/en/blog/2021/12/vulnerability-notes-log4shell/.

[17] *Sfruttamento della CVE-2021-44228 riguardante Apache Log4j (BL01/211212/CSIRT-ITA)*. 2021. URL: https://www.csirt.gov.it/contenuti/rilasciato-poc-pubblico-per-lo-sfruttamento-della-cve-2021-44228-che-riguarda-apache-log4j-bl01-211212-csirt-ita.

[18] *Mitigating Log4Shell and Other Log4j-Related Vulnerabilities*. 2021. URL: https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-356a.

[19] *FTC warns companies to remediate Log4j security vulnerability*. 2022. URL: https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2022/01/ftc-warns-companies-remediate-log4j-security-vulnerability.

[20] Patrick Howell O'Neill. *The internet runs on free open-source software. Who pays to fix it?* URL: https://www.technologyreview.com/2021/12/17/1042692/log4j-internet-open-source-hacking/.

[21] Kent Walker. *Why we're committing $10 billion to advance cybersecurity*. 2021. URL: https://blog.google/technology/safety-security/why-were-committing-10-billion-to-advance-cybersecurity/.

[22] Raghunandan J. *Log4j Vulnerability and the Importance of SBOM in Software Supply Chain Security*. URL: https://www.appknox.com/blog/log4j-vulnerability-and-the-importance-of-sbom-in-software-supply-chain-security.

[23] Feng Li, Howard Williams e Martin Bogle. «The 'Millennium Bug': its origin, potential impact and possible solutions». In: *International Journal of Information Management* 19.1 (1999), pp. 3–15. ISSN: 0268-4012. DOI: https://doi.org/10.1016/S0268-4012(98)00043-7. URL: https://www.sciencedirect.com/science/article/pii/S0268401298000437.

[24] George Cho, Eugene Clark e Arthur Hoyle. «The Y2K Legal Problem: Legal Risk Analysis, Prevention and Coping Strategies for THE Y2K PROBLEM: LEGAL RISK ANALYSIS, PREVENTION AND COPING STRATEGIES FOR MANAGERS». In: (ott. 2011).

[25] *What Is The Year 2038 Problem?* URL: https://theyear2038problem.com/.

[26] Henrico Dolfing. *The $440 Million Software Error at Knight Capital*. URL: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html.

[27] N.G. Leveson e C.S. Turner. «An investigation of the Therac-25 accidents». In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[28] Nancy G. Leveson. «Medical Devices: the Therac-25». In: 1985. URL: https://api.semanticscholar.org/CorpusID:13148715.

[29] Gregory Travis. *How the Boeing 737 Max Disaster Looks to a Software Developer*. 2019. URL: https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer.

[30] Jeremy Bogaisky. *Boeing's Financial Toll From 737 MAX Crisis Doubles To $18.4 Billion As It Books First Full-Year Loss In 22 Years*. 2020. URL: https://www.forbes.com/sites/jeremybogaisky/2020/01/29/boeings-financial-toll-from-737-max-crisis-doubles-to-184-billion-as-it-books-full-year-loss/.

[31] Il Post. *Boeing si è dichiarata colpevole nell'indagine su due gravi incidenti aerei ai 737 Max*. 2024. URL: https://www.ilpost.it/2024/07/08/boeing-colpevole-frode-indagine-incidenti/.

[32] *Cyber Resilience Act and CE mark for digital products connected to the internet.* URL: https://h-on.it/cyber-resilience-act-and-ce-mark-for-digital-products-connected-to-the-internet/.