ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI SCIENZE

CORSO DI LAUREA TRIENNALE IN INGEGNERIA E SCIENZE INFORMATICHE

VULNERABILITY ASSESSMENT:

STUDIO DELLE FINESTRE DI VULNERABILITÀ E SVILUPPO DI UN NUOVO STRUMENTO

Tesi in

Programmazione di reti

Relatore:
D'Angelo Gabriele

Presentata da:
Luffarelli Marta

Sessione II
Anno Accademico 2017/2018

..ma dove vai, ma dove corri
a testa bassa da diecimila anni?
A che ti serve,
che ci fai col tuo progresso
se non sai più toccare
la terra e i fiori con le mani?
Sull'astronave blu
non ti dimenticare mai,
uomo,
non ci sei soltanto tu!

Indice

Indice delle figure

Introduzione

Alla base di questo studio vi è l'analisi delle vulnerabilità legate a un software e al loro ciclo di vita. In particolare, si pone l'attenzione su quanto sia inevitabile avere un software vulnerabile installato nei propri sistemi e su quanto sia quindi importante avere consapevolezza dei suoi punti deboli per non farsi cogliere impreparati a fronte di un attacco.
Le motivazioni che mi hanno spinto ad approfondire tale tema sono da imputare innanzitutto alla curiosità che l'argomento *Sicurezza* in generale mi suscita e, in secondo luogo, alle esperienze fatte in questi tre anni di Università che hanno incentivato l'interesse per tali argomenti.
L'obiettivo di questa tesi è quello di definire quali siano le fasi di vita di una vulnerabilità, analizzandone peculiarità e criticità. Il progetto, inoltre, mira a proporre una strumentazione il più possibile idonea alla raccolta di informazioni circa le vulnerabilità associate alle applicazioni.
È stato condotto uno studio preventivo che fornisse indicazioni circa lo stato dell'arte di questi argomenti.
La tesi è articolata in cinque capitoli: nel primo vengono fornite le descrizioni circa il significato di una vulnerabilità per un software, il significato di una finestra di vulnerabilità e di quali siano le fasi del ciclo di vita di questa finestra e, inoltre, si lasciano spunti per riflessioni che vanno oltre il contesto informatico. Nel secondo capitolo si definisce il concetto di *"Vulnerability assessment"* sottolineando come sia importante la presenza di fonti attendibili presso cui cercare informazioni. Il terzo capitolo introduce il lettore a quello che sarà il software creato per questa tesi elencando quali requisiti tale software deve possedere e quali sono le possibili fonti da cui raccogliere

informazioni. Nel quarto capitolo si descrive la fase di design del progetto, dettagliandone la struttura architetturale. Il quinto e ultimo capitolo presenta la fase di implementazione del progetto, corredata da frammenti di codice e diagrammi di vario genere.
Grazie a questo lavoro di ricerca è stato possibile analizzare il problema delle vulnerabilità di un software nelle sue varie sfaccettature e realizzare un applicativo che aiuti a prendere consapevolezza di tale problema. Tali risultati saranno esposti dettagliatamente nelle conclusioni finali di questa tesi.

Capitolo 1

Definizione del problema

Il processo di sviluppo di un software, per quanto possa essere scrupoloso e ben strutturato, ha come esito la realizzazione di un programma auspicabilmente funzionante, ma inevitabilmente *vulnerabile*.

1.1 Vulnerabilità: definizione

Ci sono varie definizioni di "vulnerabilità", accezioni che in realtà non si discostano molto le une dalle altre. Una vulnerabilità in ambito informatico può essere definita come *"a flaw in information technology product that could allow violations of security policy"* e anche: *"a technological flaw [..] that has security or survivability implications"* [1].
Il vincolo sostanziale da tenere a mente quando si vuole identificare una vulnerabilità informatica, similmente a ciò che accade nella realtà, è la soggettività: ciò che per qualcuno (ad esempio l'utente del software o lo sviluppatore) è un difetto, per altri (ad esempio un attaccante) può facilmente essere un'opportunità. Lo sviluppo o l'aggiornamento di un software, porteranno inevitabilmente a nuove vulnerabilità.
Ogni nuova vulnerabilità crea una finestra di esposizione [2] che è da intendersi come il lasso temporale che intercorre tra la nascita della vulnerabilità e la completa "cura" di ogni terminale che sfrutti software "infetto".

1.1.1 Ciclo di vita delle vulnerabilità

Analizzando le finestre di vulnerabilità da un punto di vista meramente procedurale, si possono riscontrare diverse fasi evolutive [1], schematizzate nei grafici che seguono:

- **nascita**, questa è la fase in cui viene creata, spesso involontariamente nell'ambito dello sviluppo di un software, la vulnerabilità;

- **scoperta**, può avvenire in vari modi, anche per caso, e anche ad opera di soggetti malevoli. Chi scopre la vulnerabilità non è vincolato in nessun modo a rendere pubblica la cosa;

- **pubblicazione**, nel momento in cui qualcuno scopre la vulnerabilità (non è detto sia il primo) e decide di comunicarla ad un pubblico più ampio la vulnerabilità passa in uno stato di *disclosure*. La scoperta può essere pubblicata su apposite liste (es. Bugtraq) o comunicata direttamente allo sviluppatore;

- **correzione**, nel momento in cui lo sviluppatore rilascia una versione del software che corregga il problema, la vulnerabilità passa in uno stato di correzione;

- **diffusione**, avviene quando si perde il controllo su chi sia a conoscenza della vulnerabilità;

- **scripting**, questa è la fase in cui si è riusciti ad automatizzare il processo di *exploit* della vulnerabilità, ad opera di *script* o comunque tramite istruzioni dettagliate: a questo punto, esperti e non, riescono a compromettere sistemi sfruttando la vulnerabilità;

- **morte**, questa è l'ultima fase del ciclo di vita di una vulnerabilità, ossia quando il numero di sistemi che possono essere compromessi decresce fino a livelli insignificanti, auguratamente nulli.

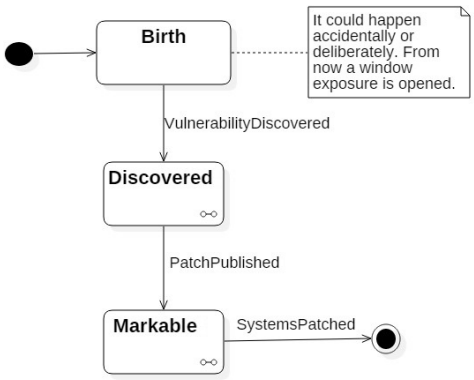


Figura 1.1: diagramma degli stati di una finestra di vulnerabilità

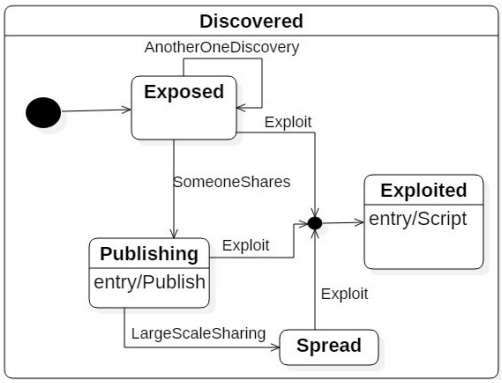


Figura 1.2: stato composito: *Discovered*

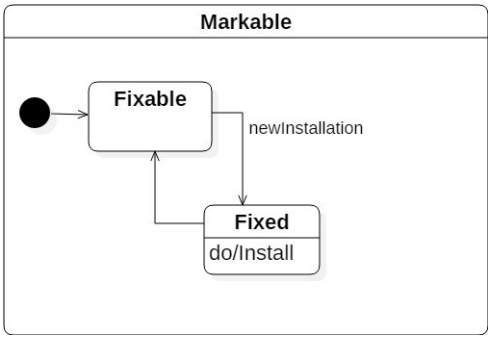


Figura 1.3: stato composito: *Markable*

Il ciclo vitale appena proposto non può essere altro che una semplificazione di quelle che sono le evoluzioni di una vulnerabilità.
Ad esempio, qualora lo sviluppo di una vulnerabilità fosse intenzionale, nascita e scoperta verrebbero a coincidere e, analogamente, qualora lo scopritore rendesse nota la vulnerabilità trovata, scoperta e pubblicazione verrebbero a coincidere.
È interessante notare come l'ultima fase, che sarebbe forse la più rilevante (perlomeno lato utente), spesso non avviene che dopo anni o non avviene affatto: ad esempio se gli sviluppatori non sono interessati a rilasciare *patch* o se è impossibile risolvere un determinato problema, o ancora se potrebbero essere state rilasciate delle versioni risolutive, ma chi gestisce il sistema non si preoccupa di aggiornare il software e installarle, ecc.

1.2 Spunti per suddivisione e approccio

La suddivisione del ciclo delle vulnerabilità per fasi non è che una possibile suddivisione.
Si potrebbe pensare ad una suddivisione su livelli, più che per fasi: il livello del software in sé e quello delle installazioni.
Si potrebbe poi ragionare su uno sdoppiamento delle finestre di vulnerabilità: una reale, che è indipendente da quando si scopre la vulnerabilità o da chi la scopre, e una che invece ne dipende.
Queste due finestre tenderanno ad allinearsi all'atto della disclosure e a ridistanziarsi nel momento in cui verrà rilasciata una patch, momento in cui la finestra "reale" sarà chiusa, mentre l'altra potrà rimanere aperta tendenzialmente all'infinito (in quanto non si potranno controllare tutte le installazioni non ancora patchate, a meno di aggiornamenti obbligatori) o essere considerata a sua volta come composita di N diverse finestre virtuali, tante quante sono le distribuzioni del software in circolazione non ancora "fixate".
L'analisi della struttura delle vulnerabilità e il loro impatto nella società tecnologica di oggi fanno pensare a diversi modi di affrontare il problema.

Bruce Schneier in "Secrets and Lies" analizza il problema sotto un nuovo punto di vista: dato che è impossibile non avere vulnerabilità e data la difficoltà nel chiudere una finestra di vulnerabilità, è bene sviluppare sistemi resistenti alle vulnerabilità, che le mettano in conto e che le riescano a gestire mediante *detection* e*response*. A suo parere, un sistema siffatto sarà in grado di riconoscere un attacco in corso e chiudere completamente la finestra di esposizione. [2]

1.3 Aspetti etico-legali

L'obiettivo primario di chi lavora in questo ambito è comune: rendere le finestre di vulnerabilità più piccole possibile, nello spazio o nel tempo [2]. Sebbene lo scopo ultimo sia lo stesso, i modi per raggiungerlo sono spesso del tutto antitetici e dibattuti.
Sin dalle prime controversie in ambito di vulnerabilità si è sempre discusso circa l'implicazione che abbia la fase di disclosure nella proliferazione di script atti a "bucare" i sistemi.
L'opinione di molti mette in chiaro il fatto che rendere di pubblico dominio l'esistenza di una vulnerabilità aumenti le probabilità di un intervento malevolo e quindi, secondo loro, sarebbe bene pubblicare poco o niente e anzi, rendere illegale qualsiasi programma per attacchi, e perseguibile il suo creatore.
Dall'altro lato del contenzioso ci sono coloro che prediligono la diffusione su larga scala di quante più informazioni si possono dare circa una vulnerabilità, in modo tale da forzare gli sviluppatori a trovare una soluzione in tempi stretti. Sebbene entrambe le soluzioni sembrino valide in teoria, la pratica è ben diversa: è praticamente impossibile controllare la diffusione delle notizie circa una vulnerabilità ed è praticamente impossibile essere sicuri che gli sviluppatori siano realmente interessati a risolvere il problema in tempi brevi e, anche qualora lo facessero, è praticamente impossibile avere la certezza che tutti si informino e aggiornino i loro software alla patch più recente.

Oltre a considerare aspetti tecnici, bisognerebbe valutare anche l'impatto morale e legale che avrebbe una scelta piuttosto che un'altra: si può davvero perseguire legalmente chi scrive software in grado di rilevare una vulnerabilità?
Si può davvero disseminare il Web di notizie circa una vulnerabilità senza tenere conto che i diretti interessati (gli utilizzatori) verranno esposti inevitabilmente ad attacchi?

1.4 Grafici di esempio

Descrivere e graficare l'evoluzione di una finestra di vulnerabilità non è semplice, soprattutto perché, per ogni vulnerabilità, ci sono dati diversi o addirittura non ci sono dati che possano aiutare ad estrapolare una curva coerente per tutti i casi.
L'idea sorta è quella di rappresentare un grafico come segue:

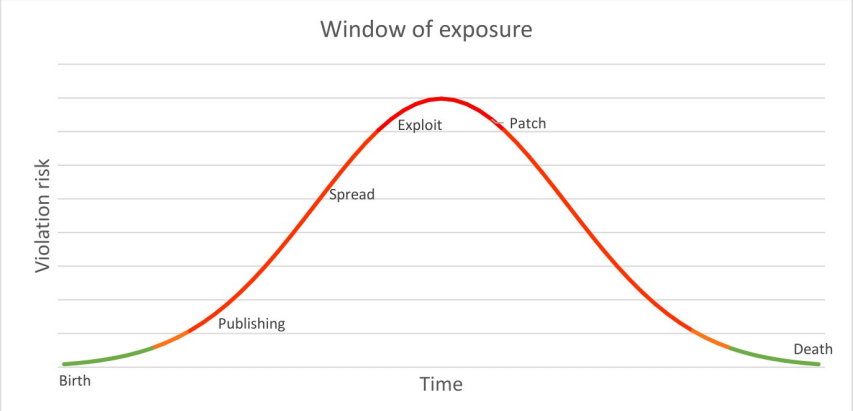


Figura 1.4: il grafico riporta l'evoluzione di una finestra di vulnerabilità. Sull'asse orizzontale è misurato il tempo, su quello verticale il rischio che un sistema vulnerabile venga violato. Si noti, come indicato anche dai colori, che un sistema è maggiormente a rischio dal momento in cui nascono degli script in grado di sfruttare la vulnerabilità in studio.

La curva, sebbene approssimi in maniera coerente il susseguirsi delle fasi di una finestra di vulnerabilità, non è derivata da dati strettamente empirici e tende a degenerare in altre sotto curve a seconda che le varie fasi abbiano o meno luogo e quando.
Di seguito alcuni esempi di cambiamenti nella curva.

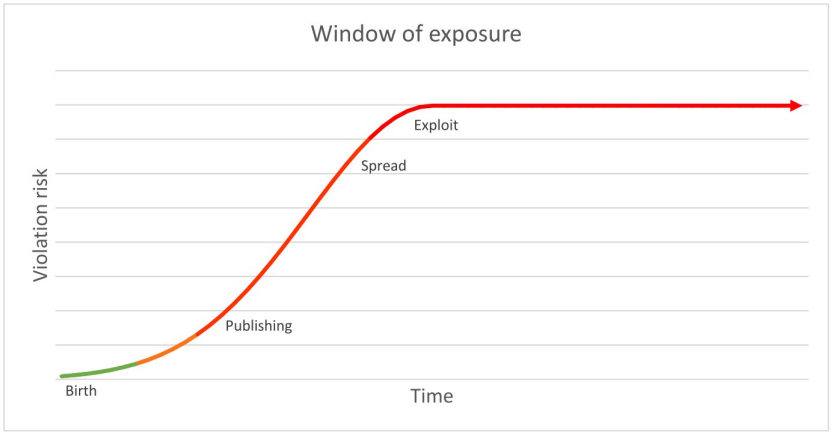


Figura 1.5: qualora i progettisti del sistema vulnerabile non corrreggessero la vulnerabilità non rilasciando alcuna patch i sistemi rimarrebbero perennemente in stato di massimo rischio.

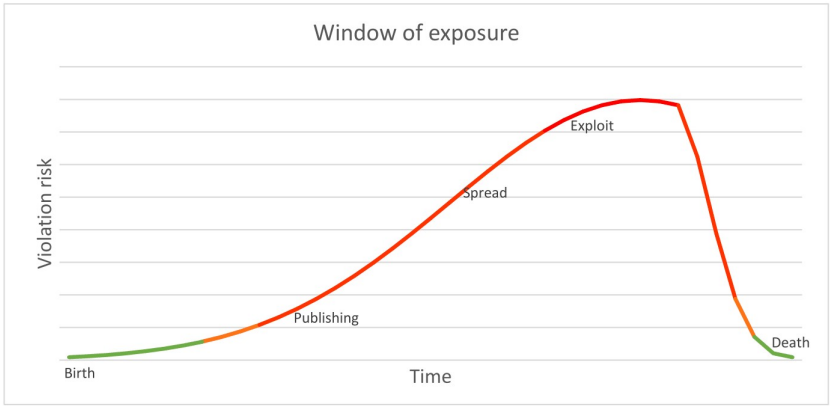


Figura 1.6: se i progettisti del sistema vulnerabile non solo provvedessero a correggere la vulnerabilità e a rilasciare una patch, ma pubblicassero anche un aggiornamento obbligatorio per installarla, allora i sistemi vulnerabili tenderebbero a diminuire quasi istantaneamente, portando rapidamente alla chiusura della finestra di vulnerabilità.

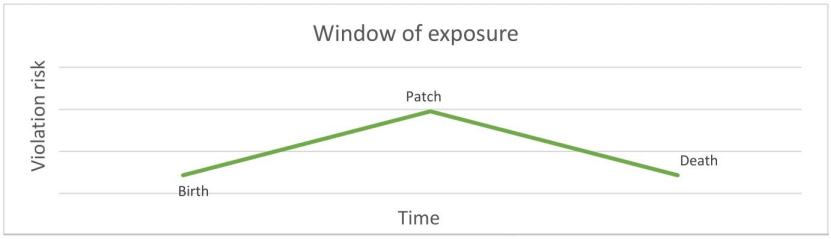


Figura 1.7: questo è il caso in cui già a livello di progettazione di un sistema ci si accorge di una "falla" e si provvede a correggerla.

1.4.1 Analisi del grafico

Il grafico in figura 1.4 è suddivisibile in due parti: tracciando una linea verticale al centro del grafico possiamo notare che nella parte sinistra si fa sì riferimento al grado di rischio dei sistemi vulnerabili, ma sulla base di quanto è nota la vulnerabilità (più si parla della cosa, più si è interessati a sfruttare la vulnerabilità); nella parte destra, invece, si fa strettamente riferimento alla velocità in cui i singoli sistemi installano correttamente una patch e quindi alla cardinalità decrescente dei sistemi esposti.
A fronte di questa suddivisione si può pensare di prendere alcuni dati reali da cui estrapolare la curva.
In particolare:

- *per la parte sinistra*, si possono usare motori di ricerca come Google Trends o social network come Twitter che, grazie a funzionalità aggiuntive come *Twitonomy*, riesce a rilevare e graficare gli *hashtag* più *twittare*;

- *per la parte destra*, si potrebbe pensare di utilizzare dati estrapolati da alcuni applicativi in grado di analizzare i sistemi di una rete (*Manage Engine Desktop Central*).

Capitolo 2

Vulnerability assessment

La proliferazione di vulnerabilità associate ai sistemi informatici ha reso opportuno sviluppare metodologie pratiche che individuassero e analizzassero queste vulnerabilità.

2.1 Definizione

Il termine *vulnerability assessment*, in informatica, indica il processo di definizione, identificazione, classificazione e priorizzazione delle vulnerabilità potenziali dei sistemi e delle applicazioni. Questo processo, di solito, prevede l'uso di tool di test automatici i cui risultati sono presentati all'utente sotto forma di report[6].
Valutare i rischi a cui è sottoposto un sistema è fondamentale non solo per provare a risolvere la falla, ma anche e soprattutto per avere piena consapevolezza di quali sono i punti deboli ed essere pronti a reagire in caso di attacco.

2.2 Tipologie

A seconda del tipo di applicativo possono essere necessari più scan, ognuno specializzato nella ricerca di un tipo di vulnerabilità.

Alcuni di questi scan sono:

- *wireless network scan*, usato per individuare i punti deboli dell'infrastruttura di rete wireless;
- *database scan*, usato per individuare i punti deboli in una base di dati;
- *application scan*, usato per scandagliare siti web alla ricerca di vulnerabilità note per un software [6];

2.2.1 Application scan: fonti

La ricerca di vulnerabilità passa per fonti più o meno attendibili e più o meno aggiornate. Spesso, infatti, le fonti più aggiornate sono quelle più ufficiose e quindi tendenzialmente meno attendibili. È importante non fare totale affidamento sull'esito di uno scan in quanto la ricerca che si effettua potrebbe essere settoriale o poco approfondita e, in generale, la fonte interrogata potrebbe non essere a conoscenza di vulnerabilità per il software in analisi.

2.2.2 National Vulnerability Database

Il *National Vulnerability Database* (*NVD*), nato nel 2005, è un repository contenente le informazioni necessarie per agevolare la ricerca e la gestione delle vulnerabilità note. La lista prodotta da NVD non è da considerarsi esaustiva, in quanto contiene solo un sottoinsieme delle vulnerabilità che possono affliggere un software. NVD si basa sulla definizione di tre concetti cardine:

- *Common Weakness Enumeration* (*CWE*), è una lista creata collettivamente e scaricabile in diversi formati. Il file CWE assegna ad ogni vulnerabilità un codice univoco, CWE-ID, e altre informazioni come, per esempio, una categoria e una descrizione;

– *Common Platform Enumeration* (*CPE*), è uno schema scaricabile e strutturato per sistemi informatici, software e pacchetti. Ad ogni prodotto individuato viene assegnato un CPE-ID e altre informazioni come, ad esempio, una descrizione;

– *Common Vulnerabilities and Exposures* (*CVE*), è un elenco scaricabile le cui voci hanno un ID univoco e, tra le altre informazioni, contengono quelle atte a collegare uno o più CPE-ID ad un CWE-ID.

Capitolo 3

Requisiti del software

Il nuovo strumento realizzato nel progetto di questa tesi, utile per la ricerca di vulnerabilità associate ad un software, è l'atto concreto di un'idea più ampia e generica.

3.1 Analisi

Il software da realizzare deve fornire la possibilità di individuare le vulnerabilità associate alla versione di uno o più software. L'utente avrà la possibilità di scegliere in quale modalità inserire il software da analizzare:

- *manuale*, potrà ricercare un programma per volta, inserendone nome e versione di interesse;
- *automatica*, verrà mostrata una lista contenente i programmi attualmente installati sul computer tra cui potrà scegliere quali analizzare (anche tutti).

Una volta individuato cosa esaminare, si sceglierà la fonte presso cui prendere le informazioni. All'utente verrà mostrata una lista delle fonti disponibili e la ricerca verrà effettuata su quelle selezionate.
Al termine delle ricerche si otterranno i risultati, suddivisi per i vari software e le varie fonti selezionati, e l'utente potrà effettuare una nuova ricerca.

Il diagramma in figura 3.1 sintetizza le funzionalità che dovrà avere il software, enfatizzando il ruolo che hanno le fonti. Esse, infatti, agiranno attivamente per permettere all'utente di ricevere risposte esaustive e aggiornate.

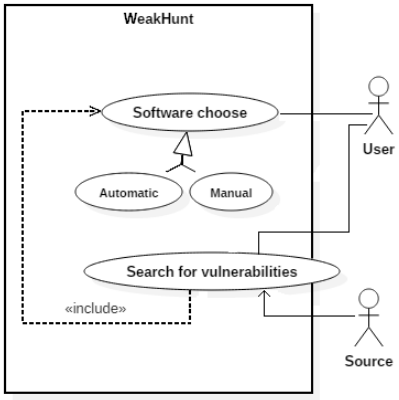


Figura 3.1: diagramma dei casi d'uso del software da realizzare

Lo schema in figura 3.2 descrive come sarà la struttura statica del sistema in termini di classi e loro relazioni reciproche, astraendo il dominio del software ed escludendo i dettagli implementativi, tipici della fase di progettazione.

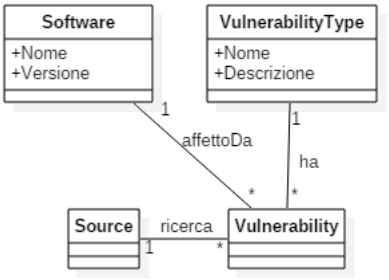


Figura 3.2: diagramma delle classi - analisi

3.2 Fonti

Le possibili fonti di notizie circa le vulnerabilità di un software sono di vario genere e più o meno affidabili/ufficiali. L'idea è quella di predisporre il software all'aggiunta di nuove fonti su cui cercare informazioni in modo da poter rendere il più possibile ampia e approfondita la ricerca.
Di seguito alcuni esempi di fonti note:

- *database*;
- *social network*;
- *canali IRC*;
- *mailing list*.

3.2.1 Database

Questo tipo di fonte è composta per la maggior parte da dei database che promuovono la pubblicazione di vulnerabilità note. Oltre il già citato NVD, ce ne sono molteplici che sfruttano gli stessi punti cardine per gestire la diffusione di notizie circa le vulnerabilità, ossia CPE, CVE, CWE:

- *CERT/CC Vulnerability Notes Database*, gestito da CERT/CC;
- *Exploit Database*, gestito da Offensive Security;
- *JVN iPedia*, gestito da IPA;
- *scip VulDB*, gestito da scip AG.

Altri database, invece, non sfruttano questi codici standard:

- *Vulnerability & Exploit Database*, gestito da Rapid7;
- *WooYun.org*, gestito da WooYun;
- *China National Vulnerability Database of Information Security*, gestito da China Information Security Evaluation Center.

Per un elenco più completo e per ulteriori informazioni sui database citati si faccia riferimento a *Vulnerability Database Catalog* [7].

3.2.2 Social network

I social network, a volte, possono essere fonti ingannevoli e fuorvianti, mentre altre possono rivelarsi alternative utili e più rapide rispetto ai database ufficiali.
Esempio calzante è *Twitter*, rete sociale creata nel 2006 dalla Obvious Corporation di San Francisco, che nel corso degli anni ha raggiunto una rilevanza tale da diventare un punto di riferimento per chiunque voglia ricevere aggiornamenti o anche richiedere assistenza su svariati argomenti.
La popolarità raggiunta da questo social network, come anche da altri, ha portato allo sviluppo di *bot* che permettessero di automatizzare alcune azioni. Implementare un bot che effettui un'iscrizione a Twitter e che ricerchi *tweet* che citano vulnerabilità e software, potrebbe essere un'ottima fonte di notizie, possibile da attuare in Java grazie alla libreria Twitter4J. Il codice di seguito riportato, tratto da *Twitter4J - code examples* [5], mostra come sia possibile effettuare una ricerca automatizzata di tweet mediante parole chiave.

```
// The factory instance is re-useable and thread safe.
Twitter twitter = TwitterFactory.getSingleton();
Query query = new Query("source:twitter4j yusukey");
QueryResult result = twitter.search(query);
for (Status status : result.getTweets()) {
System.out.println("@" + status.getUser().getScreenName() + ":" +
    status.getText());
}
```

3.2.3 Canali IRC

Internet Relay Chat (*IRC*) è un protocollo di messaggistica istantanea su Internet che consente sia la comunicazione diretta fra due utenti, sia il dialogo simultaneo di più persone divise in "stanze" detti canali. [8]
La ricerca di vulnerabilità su fonti di questo tipo può essere effettuata tramite l'implementazione di bot automatici che si sottoscrivano a canali IRC appropriati (esempio può essere quello legato alla Metasploit Community [4]) e che controllino se ci sono riferimenti al software di interesse.

3.2.4 Mailing list

Le mailing list sono delle liste di distribuzione usate da aziende, associazioni, organizzazioni o singoli per la distribuzione di informazioni utili agli iscritti alla lista.
La ricerca di vulnerabilità su fonti di questo tipo può essere effettuata tramite l'implementazione di bot automatici che si iscrivano alla mailing list di interesse (esempio può essere legato alla mailing list della Metasploit Community [4]) e ne controllino le informazioni relative al software di interesse.

Capitolo 4

Progettazione

Con il termine *progettazione* o *software design* si intende il processo che trasforma, attraverso numerosi passi intermedi, le specifiche definite in fase di analisi in un insieme di specifiche direttamente utilizzabili dal programmatore. Il risultato del processo di design è l'architettura del software, ossia l'insieme dei moduli che compongono il sistema, la descrizione della loro funzione e delle loro relazioni reciproche. Una volta conclusa la fase di analisi, fase in cui si è attenti a *cosa* deve fare il software, si passa quindi a quella di progettazione dove si dà un peso maggiore a *come* il software deve realizzare quanto richiesto. Il linguaggio di programmazione scelto per questa tesi è *Java*, linguaggio ad alto livello, orientato agli oggetti e a tipizzazione statica il cui obiettivo principale è quello di essere il più possibile indipendente dalla piattaforma di esecuzione.

4.1 Struttura architetturale

La progettazione orientata agli oggetti portata avanti in questo lavoro di tesi, si sposa perfettamente con la scelta di strutturare il software mediante l'utilizzo di uno dei più noti pattern architetturali: *Model-View-Controller* (*MVC*). I pattern architetturali esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software e *MVC* è uno dei più utilizzati.

Questo pattern, seguendo quanto scritto in *Design Patterns* [3], è caratterizzato da tre tipi di oggetti: il *Model* è l'oggetto dell'applicazione, la *View* è la sua rappresentazione grafica e il *Controller* definisce il modo in cui l'interfaccia grafica dovrà reagire a fronte di determinati input dell'utente.
La potenza di questo pattern risiede nel riuscire a disaccoppiare la *View* dal *Model*, stabilendo tra di loro un protocollo di tipo *Subscribe/Notify*. La *View* dovrà comunque assicurarsi di essere consistente con lo stato del *Model*: qualsiasi cambiamento avvenga nel *Model* verrà notificato alle *View* che ne dipendono, le quali valuteranno gli aggiornamenti da effettuare. Questo approccio permette sia di agganciare più *View* a un singolo *Model*, così da ottenere diverse rappresentazioni, sia di cambiare il modo in cui l'interfaccia grafica reagisce a fronte di un input dell'utente (possibile grazie all'incapsulamento del meccanismo di reazione in un oggetto *Controller*).
La struttura di questo pattern, applicata al caso di studio di questa tesi, è mostrata in 4.1 e verrà ampiamente commentata nel proseguo di questo capitolo.

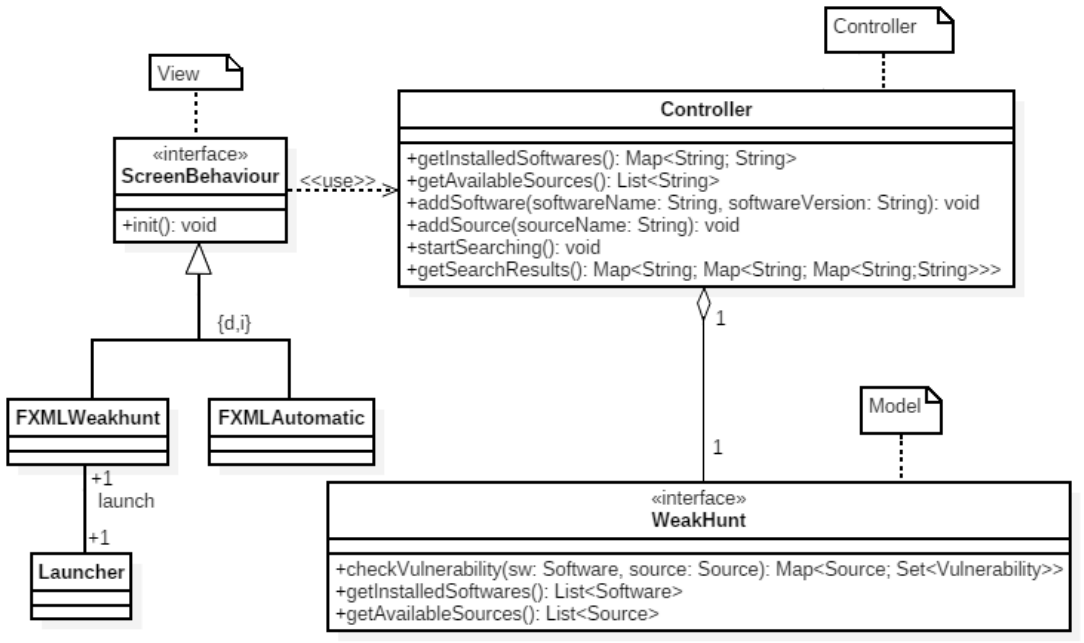


Figura 4.1: pattern MVC applicato al dominio del problema

4.1.1 Model

La struttura concepita per fungere da *Model* del sistema è sintetizzata nel diagramma in figura 4.2. Dallo schema si può osservare come la classe `WeakHunt`, entità cardine del *Model*, si relazioni con le interfacce *Source* e *AutomaticAcquisition*, rispettivamente deputate alla caratterizzazione di una generica fonte di notizie circa le vulnerabilità e alla definizione di una metodologia per ottenere la lista di software installati sul computer su cui si effettua la ricerca. Analizzando le relazioni che sussistono tra le classi `Vulnerability`, `VulnerabilityType`, `Software` e `NVDSource` si può notare che una specifica vulnerabilità esiste come entità nel dominio in relazione alla fonte che l'ha classificata e al software cui fa riferimento: si avrà un'istanza della classe `Vulnerability` diversa per ogni coppia fonte-software, anche se il tipo potrà coincidere.

Il *Model* verrà interpellato dal *Controller* nel momento in cui l'utente, tramite l'interfaccia grafica a disposizione, richiederà di visualizzare la lista dei programmi installati sul computer, piuttosto che la lista delle fonti disponibili per effettuare delle ricerche e, in definitiva, quando richiederà di iniziare effettivamente la ricerca.

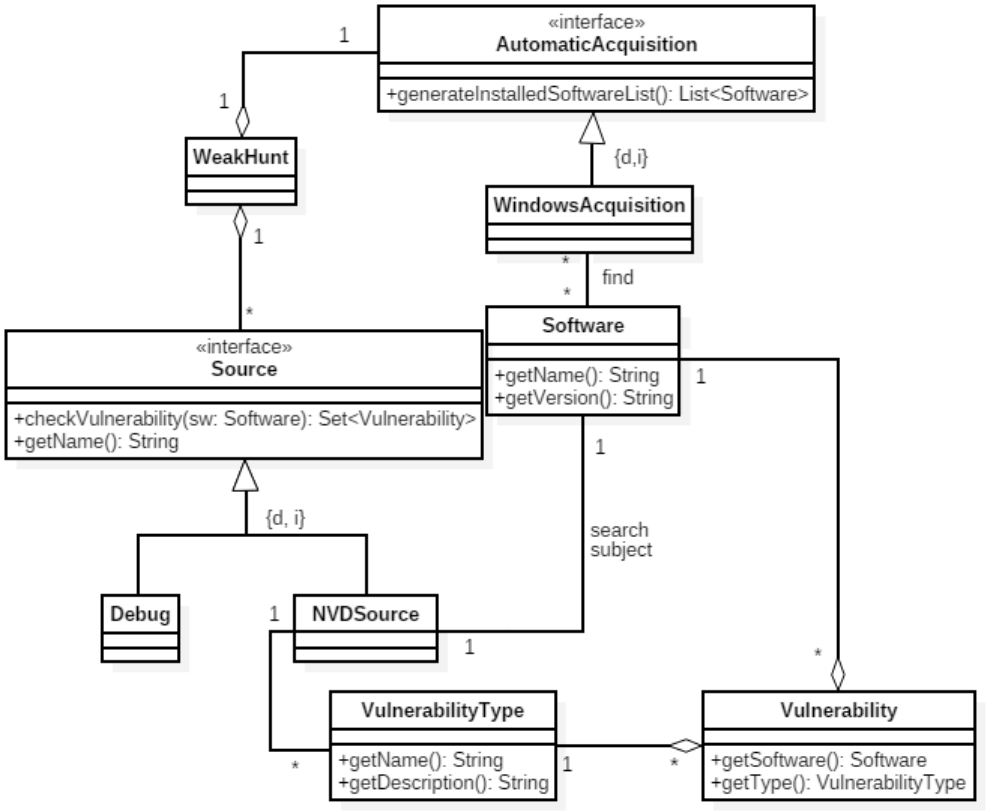


Figura 4.2: diagramma delle classi - Model

Source

L'interfaccia *Source* è stata realizzata per garantire che il software mantenesse un alto tasso di estendibilità, in modo tale che l'aggiunta di nuove fonti e la modifica di quelle esistenti avesse poco impatto sul funzionamento dell'applicazione in generale.

L'idea di base è codificata come segue:

```
/**
* A source is a place where to find software's vulnerabilities.
* It might be a database or Twitter etc.
*
*/
public interface Source {
   /**
   * This method is called to start searching a vulnerability for a
       specific software
   * @param sw
   * @return the list of the vulnerabilities found (where present)
   */
   Set<Vulnerability> checkVulnerability (Software sw);
   /**
   *
   * @return the name of the source, to inform user
   */
   String getName();
}
```

Automatic Acquisition

Uno dei requisiti richiesti all'applicazione è quello di poter ricercare vulnerabilità note associate a software che sono installati sul computer, in modo che le loro informazioni vengano presentate automaticamente all'utente. Per realizzare questa funzionalità sono stati presi in considerazione i problemi legati alla ricerca automatica dei programmi installati: ogni sistema operativo ha un proprio modo per mantenere tali informazioni e un proprio modo per mostrarle, di conseguenza si è reso necessario astrarre da quello che è il sistema operativo su cui si esegue l'applicazione, rendendo trasparente all'utente questo processo di creazione della lista dei software installati.

A livello di codice avremo la seguente situazione:

```
/**
*  This interface allows you to get installed softwares on a computer
*/
public interface AutomaticAcquisition {
   /**
   * This method will check for installed softwares and collect
       them  into a list
   * @return the list of installed softwares
   */
   List<Software> generateInstalledSoftwareList();
}
```

4.1.2 Controller

La classe `Controller` gioca il ruolo fondamentale di mettere in relazione la *View* e il *Model* e, in quest'applicazione, consiste di un'unica istanza. Per realizzare questa unicità è stato utilizzato il pattern *Singleton*, pattern creazionale che permette di avere un'unica istanza accedibile globalmente nell'applicazione. In figura 4.3 viene mostrato come il pattern *Singleton* è utilizzato nella classe `Controller`:

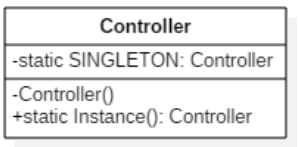


Figura 4.3: pattern Singleton per la classe Controller

La classe `Controller`, inoltre, esporrà tutti quei metodi che invocherà la *View* e che, a loro volta, invocheranno metodi del *Model* per rispondere alle richieste che l'utente invia interagendo con l'interfaccia grafica.

Uno scenario che descrive il ruolo sostanziale del *Controller* è dato dallo schema in figura 4.4 dove si mostra la sequenza di azioni che si eseguono all'avvio dell'applicazione. Il *Controller*, invocato dalla *View* tramite la classe `Launcher`, chiederà al *Model*:

- di verificare il sistema operativo su cui l'applicazione è in esecuzione prendendo come riferimento la classe opportuna per la ricerca automatica;

- di fornirgli la lista delle fonti disponibili.

Inoltre, `Controller` istanzierà `WeakHunt`, classe chiave del *Model*.

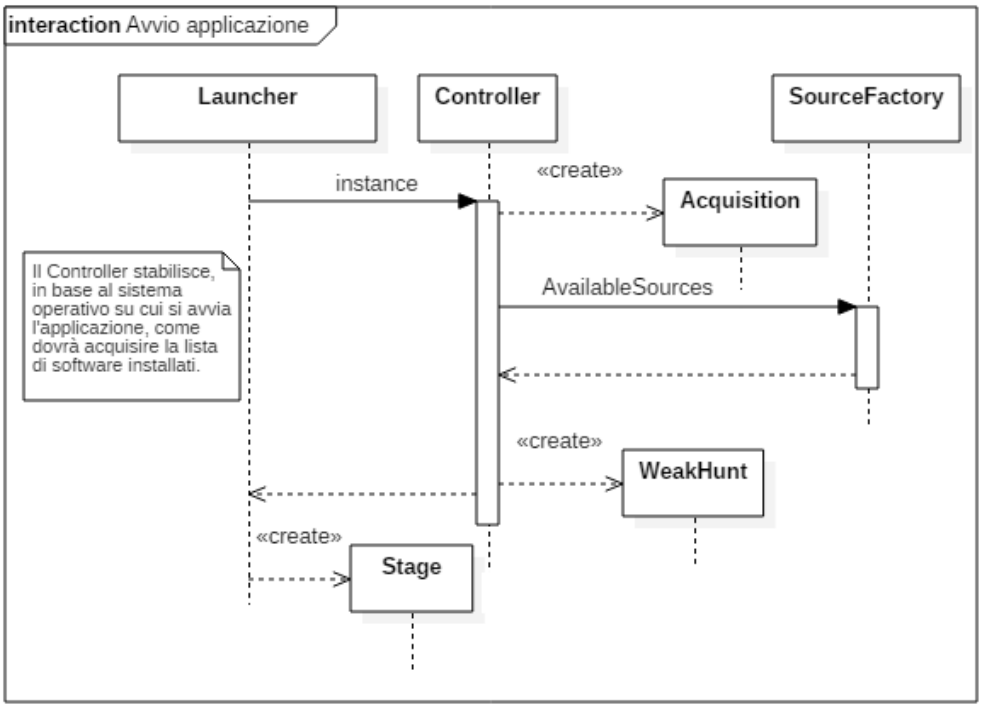


Figura 4.4: diagramma di sequenza avvio dell'applicazione

4.1.3 View

La struttura ideata per fungere da *View* è stata definita a partire dalla scelta della libreria grafica da utilizzare. I framework storicamente inclusi in *Java* sono *Swing* e *AWT* ma non offrono sempre il risultato ottimale e quindi, in questo lavoro di tesi, si è scelto di utilizzare *JavaFX*, piattaforma per interfacce utente integrata nel linguaggio a partire da *Java7*, le cui caratteristiche principali sono:

- integrazione e interoperabilità con *Swing* e *AWT* (si possono utilizzare vecchi layout e componenti già noti);
- disponibilità di FXML, un formato *XML* che permette di progettare layout in maniera simile ad *HTML* per le pagine web;
- integrazione di tecnologie di ambito web come *CSS* e *Javascript*;
- componenti di alto livello come grafici, browser, ecc.

Esiste, inoltre, un ambiente di progettazione visuale, *SceneBuilder*, che permette la composizione del codice *XML* tramite interfaccia grafica.
JavaFX si struttura a partire dall'idea di avere uno *Scene Graph*, una collezione di nodi in una struttura ad albero o grafo: lo spazio grafico di lavoro di *JavaFX* è un oggetto `Stage`, radice di ogni applicazione *JavaFX*, che può intercambiare oggetti `Scene` (contenitori di generici componenti grafici detti `Node`). `Scene` è un albero di nodi, in cui ogni nodo può essere a sua volta un contenitore o un nodo grafico. Il grafico in figura 4.5 mostra la strutturazione della *View* di questo progetto.

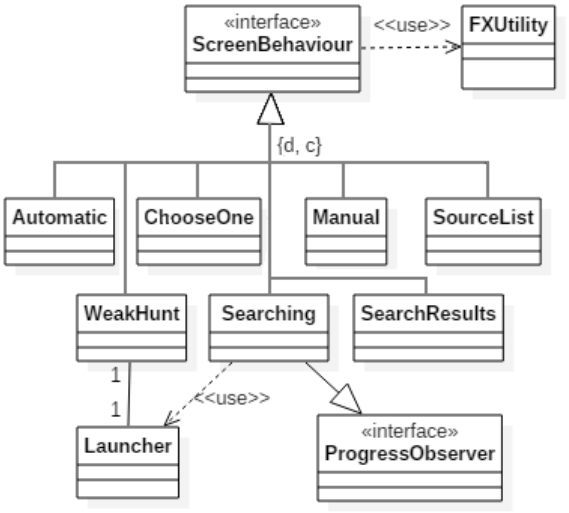


Figura 4.5: diagramma delle classi - View

Le entità principali della *View* sono `ScreenBehaviour` e `Launcher`.

Launcher

La classe `Launcher` che, come la classe `Controller`, segue il pattern *Singleton*, è la classe deputata all'avvio dell'applicazione e che verrà utilizzata dalle altre classi per effettuare cambi di scena sugli `Stage`.

ScreenBehaviour

L'interfaccia `ScreenBehaviour` descrive il comportamento delle singole schermate con i rispettivi nodi (pulsanti, label, campi per l'inserimento di valori, ecc.) esponendo il metodo `init()` che verrà richiamato nel momento in cui sarà necessario mostrare una schermata. Il funzionamento delle varie schermate è legato alla classe `FXUtility`, il cui compito è quello di cambiare scena sullo `Stage` seguendo la logica esecutiva dell'applicazione.

Come si può notare dal diagramma delle classi 4.5, le schermate utili all'applicazione sono otto:

- *WeakHunt* è la schermata che viene visualizzata all'avvio dell'applicazione e tramite cui si potrà passare alla successiva, *ChooseOne*;
- *ChooseOne* è la schermata in cui viene chiesto all'utente di scegliere in quale modalità vorrà inserire i software da analizzare. Nel caso in cui si optasse per l'inserimento manuale, si passerebbe nella schermata *Manual*, altrimenti si accederebbe alla schermata *Automatic* per scegliere i software tra quelli installati;
- *Automatic* è la schermata in cui verrà mostrata all'utente la lista di software installati tra cui potrà scegliere quali analizzati;
- *Manual* è la schermata in cui l'utente potrà inserire nome e versione del software che vuole ricercare;
- *SourceList* è la schermata in cui verrà mostrata all'utente la lista di fonti tra cui potrà scegliere quali utilizzare per ricercare vulnerabilità;
- *Searching* è la schermata in cui l'utente attende di ricevere i risultati della ricerca;
- *SearchResults* è la schermata in cui vengono proposti all'utente i risultati della ricerca.

4.1.4 Ulteriori interazioni

Molte volte, le tre parti dell'architettura del software di questo progetto comunicano tra loro per far ottenere all'utente una migliore esperienza d'uso. Ad esempio, per aggiornare l'utente sullo stato di avanzamento della ricerca nel caso in cui abbia scelto di analizzare più software contemporaneamente, è necessario controllare i risultati ottenuti dal *Controller* nel momento in cui invoca sul *Model* i metodi per iniziare la ricerca.

A questo scopo si può sfruttare il pattern *Observer*, il cui adattamento è mostrato in figura 4.6.
Il pattern *Observer* è un pattern comportamentale che definisce una dipendenza uno-a-molti tra oggetti, in modo tale che quando l'oggetto target cambia di stato, tutti gli oggetti che ne dipendono verranno notificati automaticamente. Nel caso specifico di questa tesi, `Controller` (classe target, altresì definita *Observable*) notificherà la *View* (in particolare un oggetto definito come *Observer*) non appena avrà terminato l'analisi di uno dei software da analizzare, in modo tale che l'utente osservi l'avanzamento della ricerca.

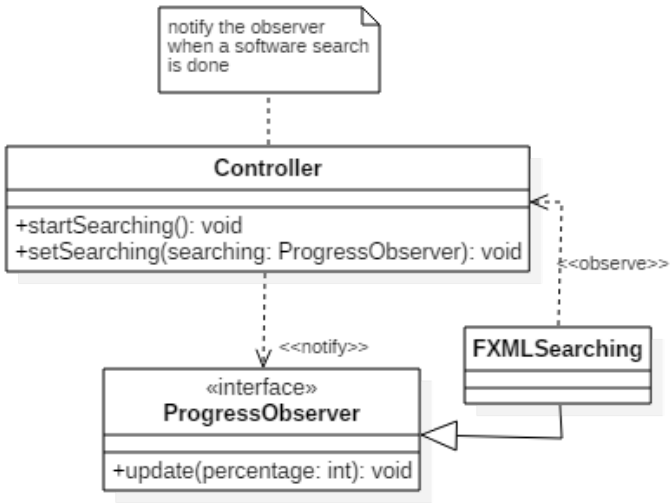


Figura 4.6: pattern Observer - Aggiornamento ricerca

Altre volte, la comunicazione tra *Model*, *View* e *Controller* avviene per completare le richieste dell'utente. Ad esempio nel momento in cui si deve rispondere alla richiesta di ricerca vulnerabilità per uno o più software su una o più fonti. Il diagramma in figura 4.7 mostra la sequenza di azioni che si innesca per poter effettuare la ricerca e mostrarla al'utente.

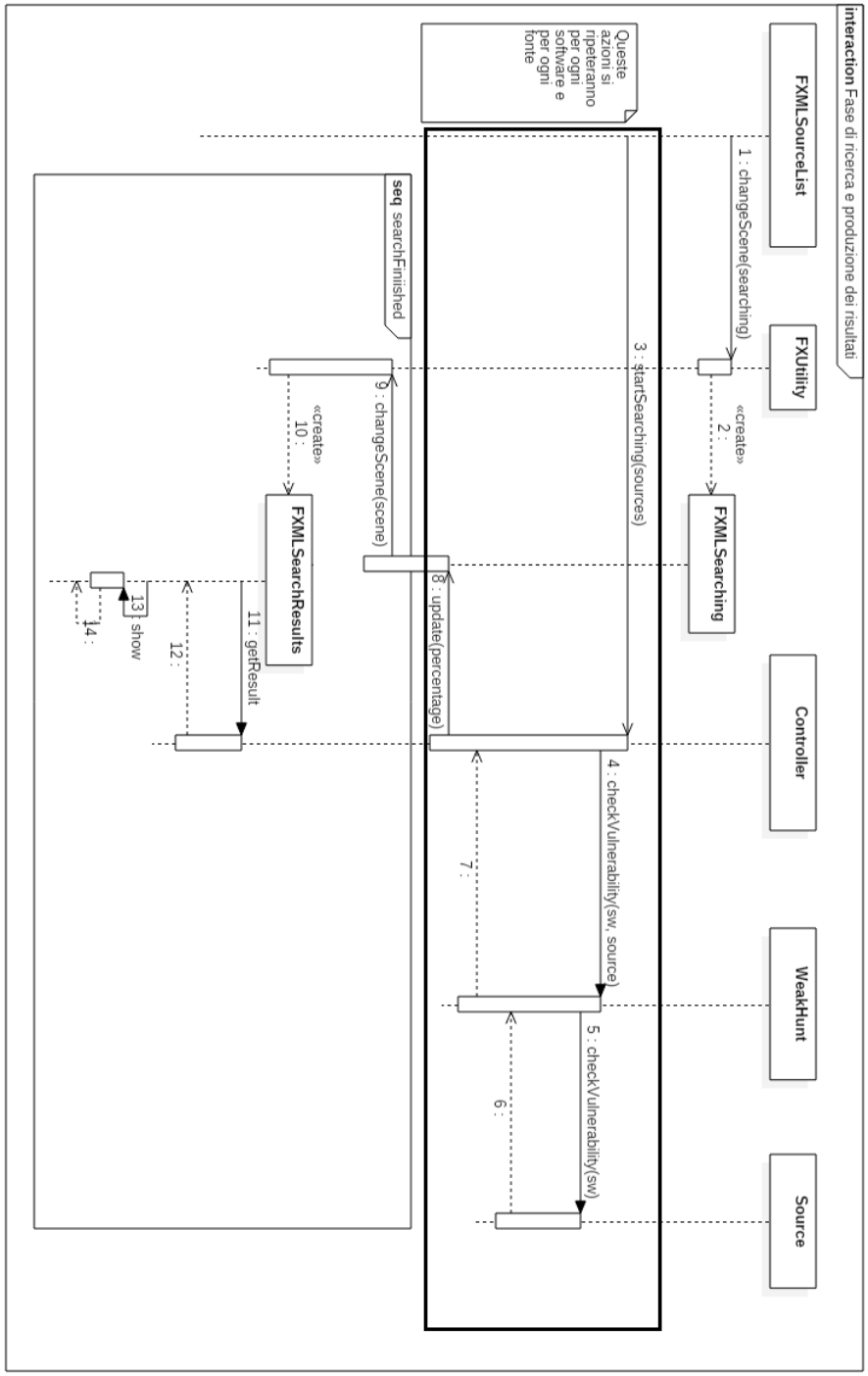


Figura 4.7: diagramma di sequenza della fase di ricerca vulnerabilità

Altre volte ancora, la comunicazione tra MVC avviene con lo scopo di mantenere un elevato tasso di modularità del codice. Ad esempio, il *Controller* non ha consapevolezza di quali o quante siano le fonti disponibili, ma fa riferimento al *Model* e in particolare alla classe `SourceFactory` per ottenerle. Questa classe, come mostrato in figura 4.8, realizza il pattern *Static Factory*, un pattern creazionale che permette di nascondere quella che è l'implementazione del concetto restituito.

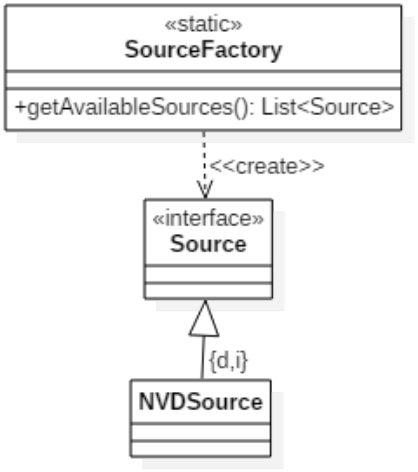


Figura 4.8: pattern Static Factory - Produzione fonti

Capitolo 5

Implementazione: Weak Hunt

L'implementazione di quanto descritto nelle fasi di analisi e progettazione ha portato ad ottenere l'applicativo *Weak Hunt*.

5.1 Dettagli implementativi

Weak Hunt offre la possibilità di ricercare le vulnerabilità unicamente presso la fonte NVD e di ottenere la lista dei software installati solamente in ambienti Windows. Affinché il software funzioni correttamente, inoltre, è opportuno avere una connessione a Internet attiva per l'aggiornamento dei file.

5.1.1 NVDSource

La classe `NVDSource`, implementazione di `Source`, si occupa di scandagliare dei file alla ricerca di corrispondenze e di aggiornarli. La fonte NVD, infatti, offre la possibilità di ricercare le informazioni necessarie analizzando una serie di file di testo scaricabili, ma, avere dei file scaricati, si traduce in un loro ripetuto scaricamento per mantenerli aggiornati. Sebbene la mole di file da scaricare e la loro grandezza potrebbe scoraggiare un loro aggiornamento, in questo progetto è stato realizzato un meccanismo che automaticamente controlli e scarichi eventuali nuovi aggiornamenti. Il meccanismo di aggiornamento è descritto nel diagramma a stati in figura 5.1.

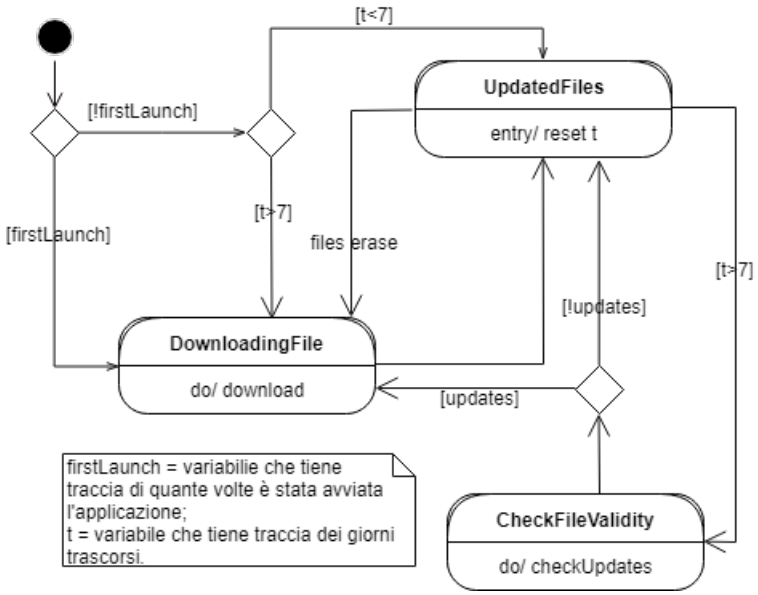


Figura 5.1: diagramma degli stati sull'aggiornamento dei file

Il fulcro della classe `NVDSource` risiede nel metodo `checkVulnerability(Software sw)` che restituisce l'insieme di vulnerabilità associate al software `sw` ed è presentato nel codice seguente:

```
public Set<Vulnerability> checkVulnerability(Software sw) {
   Optional<String> cpe = cpeSearcher.getCPE(sw);
   if(cpe.isPresent()){
      return checkVulnerabilityFiles(cpe.get(), sw);
   } else {
      return Collections.emptySet();
   }
}
```

Si noti la chiamata al metodo `checkVulnerabilityFiles(String cpe, Software sw)`, metodo privato che controlla la validità dei file (eventualmente li aggiorna tramite il metodo `mirror()` della classe `NistDataMirror`) ed effettua la ricerca del software con le dinamiche descritte nel diagramma in figura 5.2.

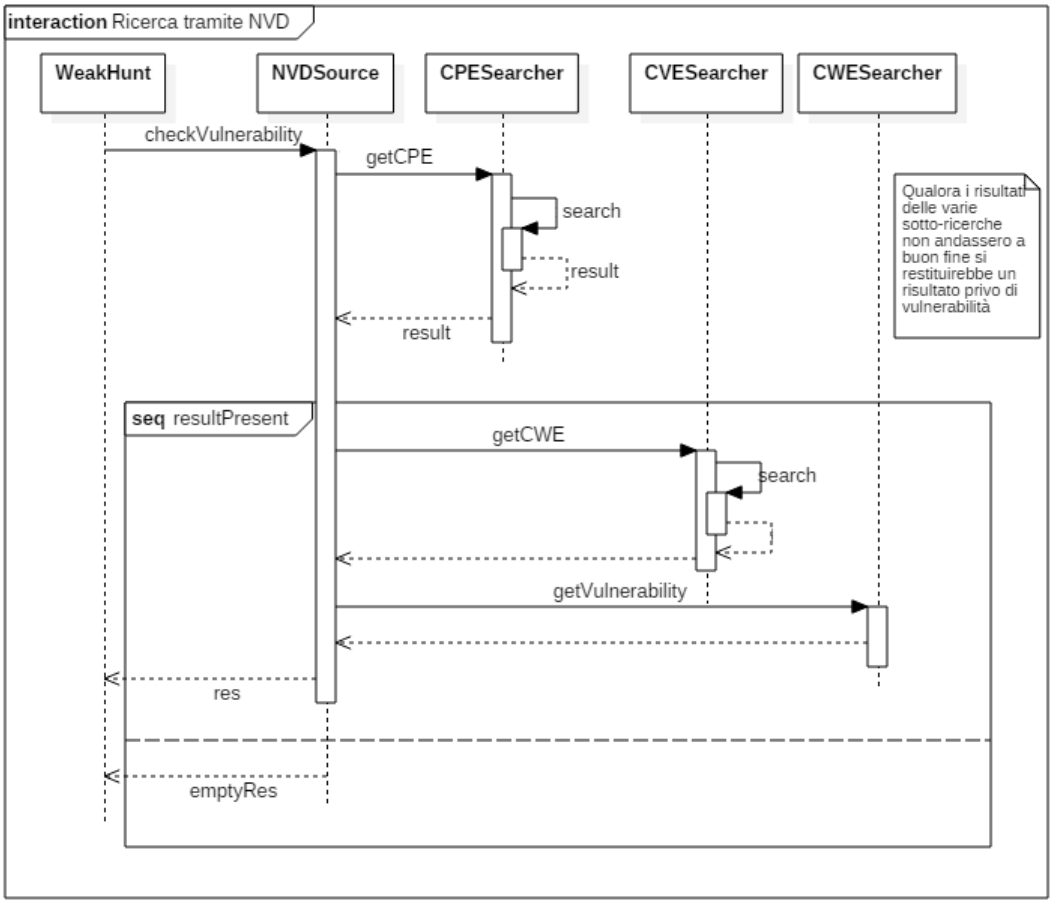


Figura 5.2: diagramma di sequenza della ricerca su NVD

Il codice del metodo `checkVulnerabilityFiles(String cpe, Software sw)` è presentato di seguito:

```
private Set<Vulnerability> checkVulnerabilityFiles(final String
    cpe, final Software sw) {
   final Set<Vulnerability> res = new HashSet<>();
   if(!checkFileValidity()) {
      NistDataMirror.mirror();
   }
   try {
      for (String fileName :
```

```
            Files.readAllLines(Paths.get(namesFile.getAbsolutePath())))
            {
           res.addAll(cweSearcher.getVulnerability(cveSearcher.getCWE(cpe,
               fileName))
           .stream().map(x -> new Vulnerability(sw,
               x)).collect(Collectors.toSet()));
        }
    } catch (IOException e) {
    return Collections.emptySet();
}
return res;
}
```

5.1.2 WindowsAcquisition

La logica dei sistemi operativi *Microsoft* è quella di conservare le opzioni e le impostazioni del sistema operativo e delle applicazioni installate in un database definito *registro di sistema*. Il registro è organizzato in una gerarchia, originata da alcune sezioni principali; ogni nodo è detto chiave (*key*) e può contenere uno o più elementi detti valori (*values*). Le chiavi di primo livello hanno il nome interamente in maiuscolo con "*HKEY*" come prefisso.
La classe `WindowsAcquisition`, che si occupa di fornire la lista dei software installati lavorando su un sistema operativo Windows, va a ricercare tali programmi nella chiave *System* (sottochiave di $HKEY\_LOCAL\_MACHINE$) in cui è memorizzata la configurazione hardware del computer.
Ottenuta la lista, gli elementi verranno filtrati tramite *regular expressions* in modo tale da ricavare nomi e versioni fruibili alle altre classi che effettueranno la ricerca.
Il codice che segue mostra il metodo `generateInstalledSoftwareList()`, invocato per ottenere la lista dei software installati.

```
public List<Software> generateInstalledSoftwareList() {
   Map<String, Software> map = new HashMap<>();
```

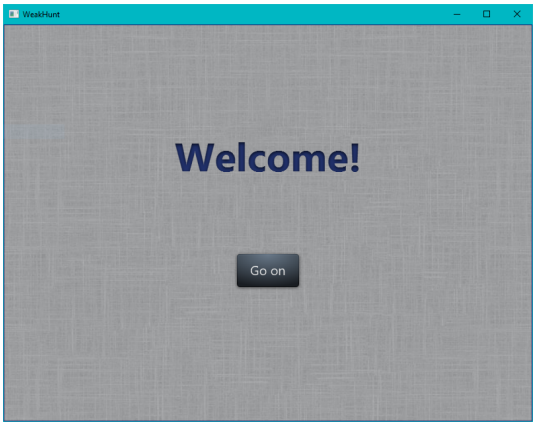
```
    final List<Software> softwares = new ArrayList<>();
    for (Paths path : Paths.values()){
        map.putAll(this.getSoftwares(path.getPath()));
    }
    for (Map.Entry<String, Software> entry : map.entrySet()){
        softwares.add(entry.getValue());
    }
    return softwares;
}
```

Questo metodo, a sua volta, si serve del metodo `getSoftwares(Path path)` che effettuerà sia la ricerca di tutti i programmi installati cercando nei percorsi indicati nell'enumerazione `Path`, sia il filtering dei risultati ottenuti a partire dal pattern mostrato di seguito:

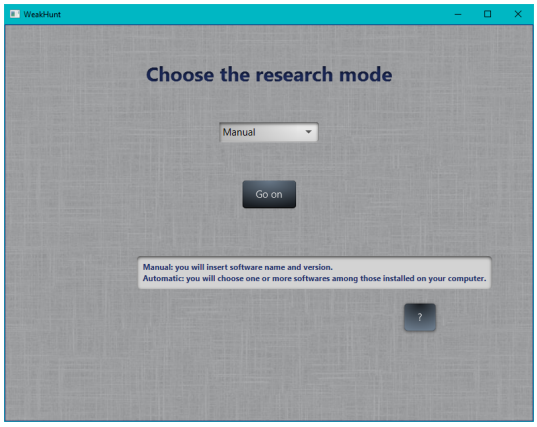
```
    private static final List<String> FILTERING_PATTERN =
        Arrays.asList("(.*)[0-9]*\\.(.*)", "(.*)-(.*)",
        "(.*)version(.*)",
    "(.*)x64(.*)", "(.*)x32(.*)", "(.*)\\((.*)", "(.*)\\)(.*)");
```

5.2 Resa grafica

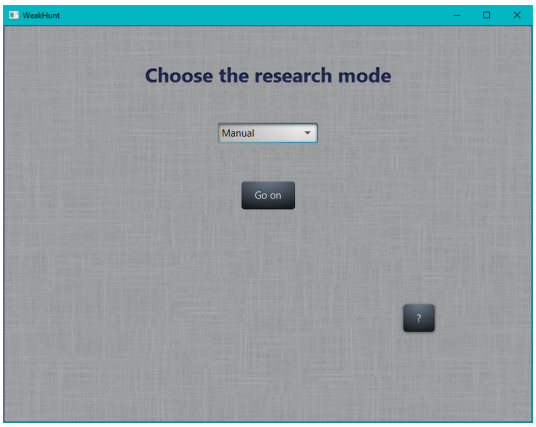
La resa grafica finale è stata realizzata con l'utilizzo di *JavaFX CSS*, basato sul *W3C CSS*, anche se non del tutto equivalente: ogni nome di proprietà in JavaFX CSS deve avere espresso il prefisso *-fx-*, anche per quelle proprietà che appaiono simili al CSS standard in quanto JavaFX assegna una semantica propria ad ogni valore. Per inserire il CSS nel software si sfrutta la proprietà dei nodi dello *Scene Graph* che permette di specificare lo stile da aggiungere ad uno specifico nodo. Il risultato ottenuto è mostrato dagli screenshot che seguono.



(a) schermata principale da cui si può accedere alla modalità di ricerca

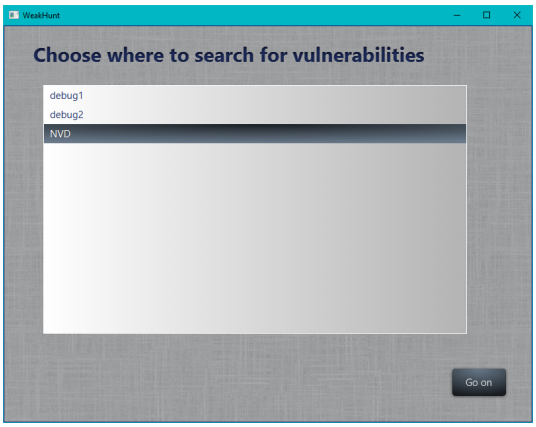


(b) schermata per scegliere la modalità di inserimento

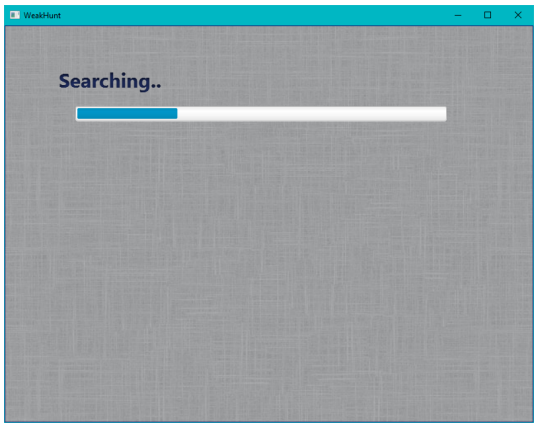


(c) Schermata che mostra la scelta "manuale"

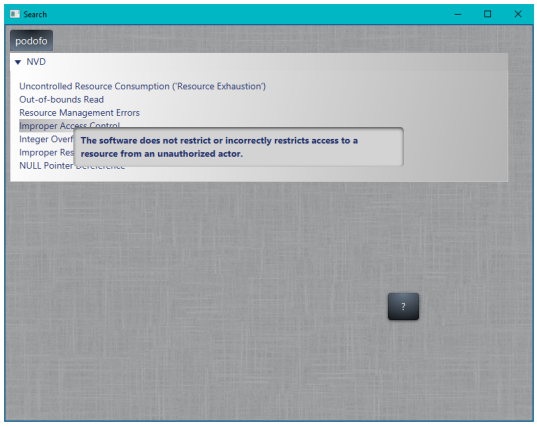
(d) Schermata per inserire nome e versione software



(e) Schermata che mostra l'elenco delle fonti disponibili

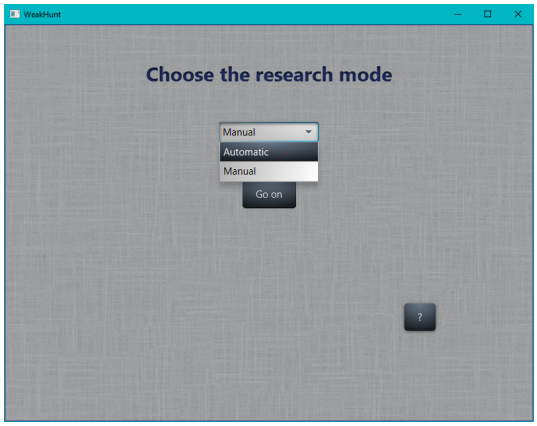


(f) Schermata che mostra l'avanzamento delle ricerche

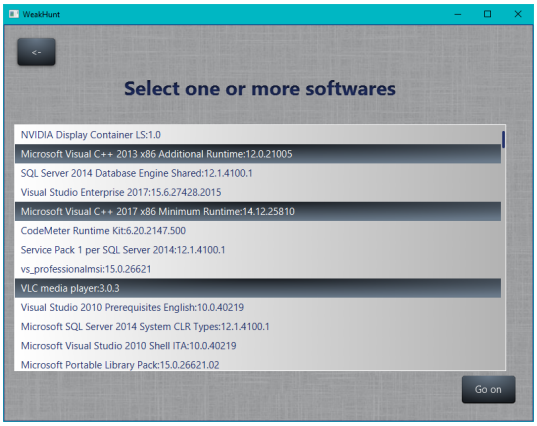


(g) Schermata che mostra i risultati della ricerca

Di seguito vengono mostrate le schermate che si vedrebbero scegliendo la modalità automatica di ricerca:



(c) Schermata che mostra la scelta "automatica"



(d) Schermata che mostra l'elenco dei software installati sul computer

Conclusioni e sviluppi futuri

Gli obiettivi principali di questa tesi, ossia portare a compimento uno studio approfondito circa le vulnerabilità legate ai software e realizzare un programma che renda l'utente medio più consapevole della moltitudine di rischi a cui si è esposti quando si ha a che fare con i software, si possono definire soddisfatti.
L'ampia trattazione affrontata nei primi capitoli esaurisce in toto quanto di nozionistico c'è da sapere sull'argomento "vulnerabilità" e lascia ampio spazio per ulteriori discussioni e approfondimenti (ad esempio, sarebbe sensato trovare metodi più rigorosi e dati concreti per graficare gli stati di una finestra di vulnerabilità).
La versione attuale del software realizzato, sebbene risponda a tutti i requisiti identificati in fase di analisi, è da considerarsi solo un punto di partenza per ulteriori modifiche e miglioramenti. Ad esempio, si potrebbe:

- migliorare le performance dell'applicazione all'atto dell'aggiornamento dei file;
- implementare altre fonti in modo da fornire una gamma di risultati più ampia e completa;
- implementare metodi di acquisizione della lista software anche per altri sistemi operativi (Linux, macOS);
- migliorare le dinamiche di ricerca delle vulnerabilità sui file NVD, migliorando i filtri di ricerca;

– aggiungere la possibilità di specificare nuovi parametri per la ricerca (ad esempio per ottenere risultati che mostrino solo le vulnerabilità non ancora risolte, o per ottenere solo vulnerabilità scoperte in un determinato gap temporale).

L'intero progetto di tesi, svolto in concomitanza con altre esperienze di studio e lavoro, mi ha reso una persona più scrupolosa e una studentessa più consapevole e realizzata per essere riuscita a mettere alla prova quanto finora studiato solo in teoria.

Acronimi

CPE Common Platform Enumeration. 12

CVE Common Vulnerabilities and Exposures. 12

CWE Common Weakness Enumeration. 12

IRC Internet Relay Chat. 15, 16

MVC Model-View-Controller. 19

NVD National Vulnerability Database. 12

Glossario

bot diminutivo di "robot", è un programma informatico che accede alla rete tramite lo stesso tipo di canali utilizzati dagli utenti umani. Programmi di questo tipo sono diffusi in relazione a diversi servizi con scopi vari ma, in genere, sono legati all'automazione di compiti che sarebbero troppo gravosi o complessi per gli utenti umani. 16

detection rilevamento, individuazione. 5

disclosure divulgazione. 2

exploit programma che sfrutta una specifica vulnerabilità presente in un sistema informatico per ottenere i privilegi di amministratore. 2

hashtag è un tipo di etichetta utilizzato in ambito web e sui social network per facilitare la ricerca di discussioni, immagini, video circa temi o contenuti specifici. 9

patch programma o file distribuito per la correzione di errori di funzionamento di un sistema o di un programma. 4

response reazione. 5

script programma che automatizza una serie di operazioni eseguite da un programma o un sistema. 2

Subscribe/Notify in generale, in questo schema, mittenti e destinatari di messaggi dialogano attraverso un tramite: il mittente di un messaggio non deve essere consapevole dell'entità dei destinatari, ma si limita a comunicare il proprio messaggio al tramite. I destinatari si rivolgono a loro volta al tramite richiedendo di essere aggiornati a ogni nuovo messaggio.. 19

tweet messaggio di testo, il cui nome è legato al social network Twitter, avente una lunghezza non superiore ai 140 caratteri (da poco aumentati a 280). 16

twittare pubblicare un tweet, breve messaggio di testo su un particolare social network (Twitter). 9

Bibliografia

[1] W. A. Arbaugh, W. L. Fithen e J. McHugh. «Windows of vulnerability: a case study analysis». In: *Computer* 33.12 (dic. 2000), pp. 52–59. ISSN: 0018-9162. DOI: 10.1109/2.889093.

[2] *Full Disclosure and the Window of Exposure*. 15 Nov. 2001. URL: https://www.schneier.com/crypto-gram/archives/2001/1115.html.

[3] Erich Gamma et al. *Design Patterns*. 1995.

[4] *Metasploit - getting support*. URL: https://metasploit.help.rapid7.com/docs/getting-support.

[5] *Twitter4J - code examples*. URL: http://twitter4j.org/en/code-examples.html.

[6] *Vulnerability assesment (vulnerability analysis)*. 2006. URL: https://searchsecurity.techtarget.com/definition/vulnerability-assessment-vulnerability-analysis.

[7] *Vulnerability Database Catalog*. 17 Mar. 2016. URL: https://www.first.org/global/sigs/vrdx/vdb-catalog.

[8] Wikipedia. *Internet Relay Chat — Wikipedia, L'enciclopedia libera*. [Online; controllata il 7-settembre-2018]. 2018. URL: https://it.wikipedia.org/wiki/Internet_Relay_Chat.