Costruire software sicuro dalle fondamenta

Antonio Parata
antonio.parata@ictsc.it

Abstract
In questo articolo si presentano le principali attività che trovano posto all'interno di un ciclo di sviluppo software security-oriented. In particolare, verranno analizzate in maggior dettaglio le attività svolte nelle prime fasi del ciclo di sviluppo software (Software Development Life Cycle - SDLC). Inoltre nel corso dell'articolo verranno esposte delle metodologie per poter integrare tali attività nel SDLC senza dover stravolgere il processo di sviluppo già esistente.

Keywords
Secure Software Development Life Cycle, Attack Tree, Attack Patterns, Attack Model, Attack Surface Analysis, Threat Modeling, Security Requirement, Security Patterns, Abuse Case, Code Auditing, Security Testing, Penetration Testing, Data Flow Diagram, Software Design, Software Architecture, Risk Evaluation.

1 INTRODUZIONE

Negli ultimi anni, l'interesse verso la software security ha visto un notevole incremento, in particolar modo per quel che riguarda la costruzione di software sicuro a partire dal ciclo di sviluppo software. Fino ad oggi l'attenzione per la sicurezza era, nella migliore delle ipotesi, limitata alla sola fase di testing. Come conseguenza, molti ricercatori di sicurezza hanno dimostrato la massiccia presenza di vulnerabilità nella maggior parte dei software costruiti seguendo questa metodologia di sviluppo. La risposta da parte delle software house è stata quella di ingaggiare professionisti nel campo della sicurezza, con il fine di trovare negli applicativi il maggior numero di vulnerabilità per poi porvi in seguito rimedio. Pur supponendo di riuscire ad identificare la maggior parte delle vulnerabilità presenti, le spese da affrontare per porvi rimedio sono spesso molto alte e difficilmente sostenibili (non è raro che la soluzione sia spesso la modifica della configurazione di un apparato di rete con il fine di riscontrare un eventuale tentativo di attacco). In **Figura 1** è rappresentato l'andamento del costo da affrontare per eliminare la vulnerabilità in base al tempo all'interno del SDLC.

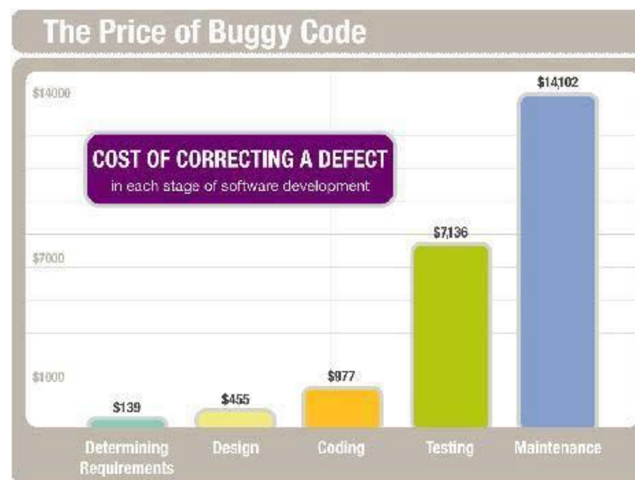

Figura 1. Diagramma di Barry Boehm, sull'andamento del costo nel rimediare alle vulnerabilità software, rispetto alla fase nel ciclo di sviluppo software.

Negli ultimi anni molte software house hanno concentrato i propri sforzi per poter includere il fattore sicurezza all'interno dei propri prodotti sin dalle prime fasi del SDLC. Il risultato è che i nuovi prodotti sviluppati risultano molto più robusti e sicuri. Un esempio di tale svolta è rappresentato da Microsoft, la quale ha adottato un nuovo ciclo di sviluppo, denominato Secure Development Life-Cycle – SDL. Il risultato è che attualmente prodotti di punta come MS SQL Server, Internet Information System e Internet Explorer 7, hanno un tasso di presenze di vulnerabilità davvero minimo (nel caso di Internet Information System addirittura nulla, alla data attuale) rispetto alle precedenti versioni o rispetto ai relativi concorrenti (sia open-source che closed-source). Nei prossimi paragrafi vedremo quali sono le attività che maggiormente influenzano la sicurezza nel ciclo di sviluppo, e in che modo integrarle nel proprio processo.

Infine nella sezione finale si presenta una ipotesi su quali saranno nel prossimo futuro le attività che maggiormente prenderanno piede all'interno del ciclo di sviluppo.

2 CICLI DI SVILUPPO SOFTWARE SECURITY ORIENTED E STANDARDS DI SICUREZZA

Esistono vari cicli di sviluppo software in letteratura, i quali si differenziano per alcune caratteristiche, come ad esempio il rigore e la formalità (si pensi ad un tipico ciclo di sviluppo a cascata rispetto ai più moderni metodi agili). Tra di essi, ve ne sono alcuni il cui compito specifico è quello di creare software sicuro. Tra quelli di maggior rilievo ci sono CLASP ([27]) e il noto ciclo di sviluppo SDL ([16,20]) re-

centemente adottato da Microsoft per lo sviluppo dei suoi prodotti di punta (per una comparazione tra questi due cicli di sviluppo si veda [14]). Entrambi adottano buona parte delle attività considerate in questo articolo. È da annoverare anche l'esistenza dei vari *Capability Maturity Model* (si veda [32,33,34]). Di particolare interesse al nostro discorso è il *Systems Security Engineering Capability Maturity Model* ([18]). Il SSE-CMM è un elenco di linee guida da applicare al proprio ciclo di sviluppo software. In base all'aderenza a tali linee guida vengono assegnate delle metriche di sicurezza anche detti livelli di maturità. Si va dal livello 0 (nessuna aderenza con il modello SSE-CMM), fino al livello 5 (massima aderenza con il modello). L'assegnamento dei suddetti livelli viene effettuato da enti esterni. Concludiamo questa rassegna accennando allo standard *Common Criteria* ([6]). Il common criteria consta di due artefatti, il *Protection Profile* e il *Security Target*. Il protection profile descrive quali sono le richieste di sicurezza del software da esaminare. A seconda di tali richieste e delle specifiche presenti nel documento del common criteria, si crea un security target, in cui sono elencate le proprietà che dovranno essere esaminate. Ovviamente tali proprietà devono soddisfare le specifiche del protection profile. Viene inoltre assegnato un *livello di valutazione* che va da EAL1 fino a EAL7. In base al livello richiesto l'esaminatore (rappresentato da un ente esterno), dovrà effettuare una serie di test più o meno approfonditi. Nonostante la sua validità, il common criteria non ha avuto una diffusa applicazione da parte delle aziende sviluppatrici di software, a causa del macchinoso processo di qualifica.

3 FASI DEL PROCESSO DI SVILUPPO

In questo articolo, per semplicità, verrà considerato un ciclo di sviluppo a cascata seguendo il modello presente in [22], va da se, che gli argomenti trattati sono applicabili a qualsiasi ciclo di sviluppo (sempre che esso preveda tali fasi). Le fasi da noi considerate sono:

1. Requisiti e casi d'uso
2. Software Design e Software Architecture
3. Pianificazione test
4. Implementazione
5. Testing e risultati
6. Feedback dall'esterno

Come già accennato, ci soffermeremo maggiormente sulle prime tre fasi, descriveremo brevemente la fase 4 e 5, e accenneremo la fase 6. In ogni fase verranno descritte le attività che prendono parte in essa per incrementare la sicurezza dell'applicazione, e in che modo possono essere integrate all'interno del proprio processo di sviluppo.

Una fase non accennata, ma di fondamentale importanza, è il training dei partecipanti alla costruzione del software. In ogni fase di sviluppo dovrebbero essere presenti dei corsi di aggiornamento obbligatori pertinenti alla sicurezza della propria area di interesse.

3.1 Requisiti e Casi d'uso

In questa fase iniziale, ci sono varie attività che possono essere svolte per creare fin dall'inizio un software sicuro, esse sono:

- Utilizzo di Attack Patterns
- Creazione di Abuse Cases
- Security Requirements

Analizziamole in ordine.

3.1.1 Attack Patterns

Cominciamo con il descrivere il ruolo ricoperto dagli Attack Patterns. Essi derivano dai design patterns ([10]), ovvero rappresentano un framework generale per riuscire ad identificare un particolare tipo di attacco. In pratica un attack pattern raggruppa le caratteristiche che hanno in comune alcune vulnerabilità. Gli attack patterns possono essere espressi in modo discorsivo (si veda [15]), oppure con un certo rigore (si veda [25]). A titolo di esempio si riporta l'attack pattern Buffer Overflow:

Buffer Overflow Attack Pattern:

Goal: Sfruttare vulnerabilità da buffer overflow per eseguire un codice malevolo sul sistema target.

Precondizioni: Un attaccante deve essere in grado di poter eseguire programmi sul sistema target.

Attacco:

AND

1. Identificare programmi eseguibili privilegiati sul sistema target che siano suscettibili a buffer overflow.
2. Creare un vettore d'attacco che conterrà il codice malevolo che si vuole far eseguire.
3. Creare il codice malevolo che si vuole far eseguire (anche detto payload).
4. Eseguire il programma in modo che prenda in input il vettore d'attacco creato al punto 2.

Postcondizioni: il payload viene eseguito sul sistema target.

Gli attack patterns vengono usati non solo nella fase di requisiti e casi d'uso, ma anche nelle successive fasi del ciclo di sviluppo ([11]). Nella fase dei requisiti, il loro compito è quello di facilitare gli analisti nella creazione degli abuse cases. Nelle restanti fasi, in modo analogo, aiutano ad identificare delle possibili vulnerabilità nel software.

3.1.2 Integrazione degli Attack Patterns nel SDLC

L'integrazione degli attack patterns è del tutto indolore all'interno del proprio ciclo di sviluppo (come lo sarà la maggior parte degli argomenti qui trattati). In questo caso non si tratta di una vera e propria attività da aggiungere, piuttosto, è un supporto all'attività di definizione degli abuse cases che verrà trattata di seguito. Per ulteriori informazioni sugli attack patterns si faccia riferimento a [4,3,2], mentre per un elenco esaustivo si veda [5].

3.1.3 Abuse Cases

Gli Abuse Cases (anche detti *Misuse Cases*) rappresentano un "tool" il cui scopo è quello di aiutare il progettista a pensare come un attaccante. Gli abuse cases, sono l'opposto dei security requirements (introdotti nella prossima sezione), ovvero il loro intento è pensare come un attaccante e cercare di creare dei possibili scenari di attacco. Per la creazione degli abuse cases, generalmente si eseguono delle sessioni di brainstorming in cui, tramite l'utilizzo di *Attack Model*, si identificano le minacce e da quest'ultime si creano gli abuse cases. Gli attack model servono per riuscire ad identificare più facilmente quali sono le varie tipologie di attacco a cui l'applicazione può andare incontro. Esempi di attack model sono: elenchi di possibili attacchi (come in [1], dove però il termine *fault model* è utilizzato come sinonimo di attack model); gli attack patterns discussi nella sezione precedente; oppure il processo STRIDE ([17]) creato da Microsoft per la classificazione delle minacce, dove per ognuna delle 5 categorie (Spoofing, Tampering, Repudation, Information Disclosure, Denial of Service e Elevation of Privilege) ci si chiede in che modo sia possibile abusarne nell'applicazione. Si noti come l'applicazione di un modello di attacco come gli attack patterns, non deve per forza essere rigido, anzi, è molto probabile che durante la stesura degli abuse cases, si identifichino nuove tipologie di attacco che andranno ad arricchire il proprio modello di attacco posseduto. Ecco dunque che un modello di attacco ben fornito può costituire per l'azienda un valore aggiunto per la costruzione di software sicuro. Di seguito un possibile esempio di abuse case ottenuto utilizzando l'attack model presente in [36]:

Un utente malintenzionato può aggirare le restrizioni sull'input che vengono applicate sul client, inviando una richiesta appositamente forgiata utilizzando tools esterni.

3.1.4 Integrazione degli Abuse Cases nel SDLC

Per quanto riguarda la loro integrazione all'interno del ciclo di sviluppo, si tratta semplicemente di aggiungere un'ulteriore fase all'interno di quella di elicitazione dei requisiti. Durante le sessioni di brainstorming per la creazione degli abuse cases, oltre ad utilizzare un modello di attacco, ci si pone delle domande sul cosa il software non deve fare. Possiamo anche pensarli come degli *anti requisiti* in cui si esprimono azioni che desideriamo il nostro software non faccia. Purtroppo, in questo caso a differenza dei normali requisiti funzionali, il campo di applicazione è troppo vasto (non è possibile esplicitare tutte le azioni che il software non deve fare), per cui in genere si restringe il campo alle azioni con impatto maggiore. È dunque buona norma classificare gli abuse cases in base alla loro pericolosità.

3.1.5 Security Requirements

I security requirements, come già accennato, rappresentano i requisiti di sicurezza del software che si sta sviluppando. Il loro utilizzo è ortogonale a quello degli abuse cases, ovvero in un requisito di sicurezza si specifica in che modo il software garantirà la propria sicurezza, o per lo meno espliciterà le azioni che sono state fatte per garantirla.

3.1.6 Integrazione dei Security Requirements nel SDLC

L'integrazione dei security requirments necessita dell'aggiunta di un'ulteriore attività nel proprio processo di sviluppo. Tipicamente, la loro stesura viene effettuata in contemporanea con gli abuse cases, ma senza mischiare le due attività. Rispetto agli abuse cases, sono più facili da elicitare, e generalmente vengono creati in modo da identificare delle soluzioni alle possibili minacce presenti negli abuse cases. Proprio a causa di questo loro legame, non è raro che si cominci con la stesura degli abuse cases, per poi intervallare l'attività di stesura dei security requirements. Concludiamo il paragrafo fornendo un esempio di security requirement derivante dall'applicazione di tecniche di defensive programming ([21]):

Tutte le richieste ricevute dal server vengono convogliate verso un unico punto di validazione dell'input, in cui vengono applicati una serie di filtri con tipologia white list.

3.2 Software Design e Software Architecture

In questa seconda fase del ciclo di sviluppo software, si comincia a definire l'architettura e il design dell'applicazione. L'architettura del software può essere pensata come un design di più alto livello. Per cui tutte le attività trattate in questa sezione possono essere applicate ad entrambe le fasi a seconda della situazione. Durante tale fase, le attività principali da svolgere sono:

- Applicazione dei Security Patterns
- Risk Analysis / Threat Modeling

Per quanto riguarda l'attività di Risk Analysis / Threat Modeling, nel campo della software security non vi è una chiara distinzione tra le due. Fondamentalmente ricoprono lo stesso ruolo, con la sola differenza che nel Threat Modeling ci si dovrebbe fermare alla sola identificazione delle minacce. In questo articolo si prenderà come definizione quella data in [35], anche se in contrasto con quanto detto in [22].

3.2.1 Security Patterns

Nei paragrafi precedenti sono stati introdotti gli attack patterns, ed è stato spiegato come usarli e in che fase del SDLC andavano applicati. Può sembrare naturale applicare i security patterns nella fase di requirements, cosi come è stato fatto per gli attack patterns, ma data la loro più stretta correlazione con i design patterns, il loro utilizzo trova impiego appunto nella fase di design. La loro definizione è identica a quella dei design patterns, con la sola differenza che il problema a cui si vuole porre rimedio è un problema di sicurezza. Per completezza viene data la definizione di security patterns presente in [34]: *"I security patterns sono un'astrazione di problemi di business che indirizzano una varietà di requisiti di sicurezza e forniscono una soluzione al problema. Essi possono essere pattern architetturali che specificano come un problema di sicurezza (security problem) possa essere risolto architetturalmente (o concettualmente), o possono essere strategie di pattern difensive tramite i quali poter scrivere codice di protezione"*.

I security patterns vengono descritti in modo simile ai design patterns, ovvero specificando

- il nome
- il problema da risolvere
- la motivazione
- la soluzione generale al problema
- la struttura della soluzione (generalmente descritta tramite diagrammi UML)
- la strategia di implementazione (ovvero i vari modi in cui il pattern può essere implementato)
- le conseguenze nell'utilizzo del security patterns
- i fattori e i rischi che devono essere considerati nell'applicazione del pattern

per una descrizione dettagliata delle singole voci si veda [34].

3.2.2 Integrazione dei Security Patterns nel SDLC

Per l'utilizzo dei security patterns si può utilizzare un approccio detto *Patterns-Driven Security Design*. Suddetto approccio può essere riassunto nelle seguenti attività:

1. Creare un'architettura candidata dell'applicazione.
2. Eseguire un analisi del rischio dell'architettura creata, identificandone le minacce. In questa fase è molto utile la consultazione degli abuse cases.
3. Cominciare a creare il design dell'applicazione tenendo conto dei requisiti di sicurezza che mitigano le minacce riscontrate.
4. Durante la creazione del design, applicare ripetutamente i security patterns conosciuti in modo che soddisfino i requisiti di sicurezza.
5. Se non esiste un security pattern che soddisfa il requisito di sicurezza, il design deve essere modificato in modo da soddisfare il requisito preposto. Una volta modificato, si estrae dal nuovo design il nuovo security pattern, che verrà inserito nel proprio catalogo per garantirne il riuso.

L'applicazione dei security patterns può essere più o meno complicata, tutto dipende dal grado di esperienza che ha l'architetto nell'utilizzo dei patterns canonici.

3.2.3 Threat Modeling

Abbiamo già detto che per i nostri scopi i termini Threat Modeling e Risk Analysis sono sinonimi. L'attività di Threat modeling è un'attività complessa nonché di fondamentale importanza. L'attività dovrebbe iniziare non appena è disponibile una prima bozza del design (o architettura) del software che si sta sviluppando. Un intero libro è stato dedicato alla realizzazione del threat modeling ([35]), per cui in questa sede ci limiteremo solo ad esporne i concetti fondamentali. I passi fondamentali per eseguire un'efficace attività di threat modeling sono rappresentati in **Figura 2**.

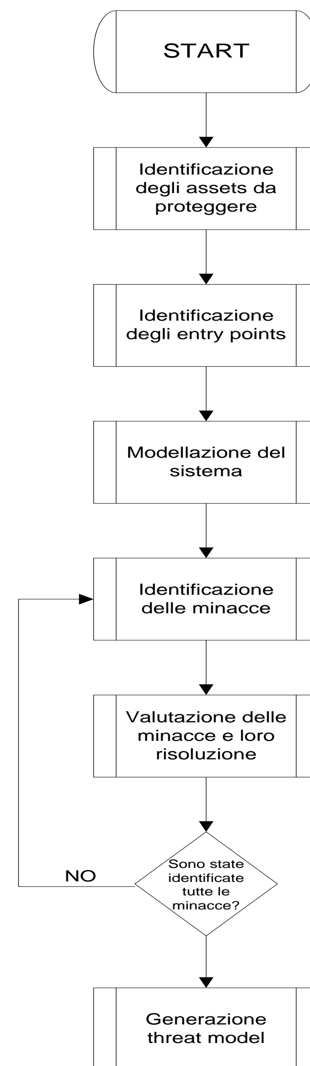

Figura 2. Diagramma delle attività del threat modeling

Commentiamo brevemente ciascuna fase:

- *Identificazione degli asset*: in questa fase vengono identificate tutte le informazioni di valore che è necessario proteggere, come ad esempio carte di credito o passwords. In questa fase vanno inseriti anche i beni di tipo astratto, come ad esempio l'immagine aziendale. Non dimentichiamo che senza beni da proteggere non esistono minacce.
- *Identificazione degli entry point*: qui vengono identificati i punti di accesso ai beni da proteggere. È importante non tralasciare nessun entry point, pena una possibile compromissione causata da un accesso lasciato incustodito.
- *Modellazione del sistema*: in questa fase si comincia a modellare il sistema. Generalmente la prassi (dettata ovviamente dal fatto che sia la migliore) è quella di creare un diagramma di flusso del sistema (in inglese data flow diagram). I diagrammi vengono creati in varie dimensioni e dettagli, a seconda della complessità del sistema. È importante far notare che non bisogna spendere troppe energie nel creare un diagramma perfetto, pena un possibile caduta in ciò che viene chiamata paralisi da analisi.
- *Identificazione delle minacce*: questa è indubbiamente tra le fasi più difficili e delicate del threat modeling. Qui viene analizzato il modello dell'applicazione e si cerca di identificare tutte le possibili minacce dirette agli assets. In questa fase vengono in genere utilizzati tutti i documenti fino ad ora creati e quelli già posseduti. Una volta identificata la minaccia, la si analizza per vedere in che modo un attaccante può portarla a termine. Un formalismo utile in questo caso è quello degli *attack trees* (anche detti *threat tree*). ([31]). Gli attack trees sono un formalismo che descrive i passi necessari a raggiungere un determinato obiettivo. Sono molto simili agli attack patterns, con la sola differenza che in questo caso non vengono specificate le mosse generiche per effettuare l'attacco, ma il tutto è contestualizzato alla situazione in esame. Inoltre un utile simbolismo grafico a forma di albero viene adottato per poter meglio analizzare la minaccia e aggiornare l'attack tree. Un esempio di attack tree è mostrato in **Figura 3**, in cui è rappresentata l'analisi della minaccia "Web Form Authentication bypass" relativa all'analisi di un'applicazione web. Il simbolo di doppia barra orizzontale sta ad indicare che entrambe le condizioni devono essere verificate affinché la condizione padre sia soddisfatta. Inoltre man mano che si procede nell'analisi e si viene a conoscenza della presenza di punti di mitigazione per particolari azioni, si può aggiornare l'attack tree inserendo dei cerchi che spiegano brevemente in che modo è stato mitigato quel pericolo, in questo modo ci si può concentrare sui passi per cui non è presente una specifica mitigazione del rischio.
- *Valutazione delle minacce e loro risoluzione*: arrivati a questo punto le minacce devono essere classificate secondo la loro importanza. Svariati modelli esistono come ad esempio DREAD (ormai considerato datato, si veda [9]), per una rassegna dei modelli esistenti si veda [30]. L'importanza di questa fase è quella di concentrarsi prima sulle minacce a più alto rischio, in modo da proporne una mitigazione. In questo modo è possibile seguire un processo di mitigazione efficiente, che cerca di eliminare le minacce a più alto rischio.
- *Generazione threat model*: infine, in output dalla attività di Threat Modeling si ha un documento con tutte le minacce riscontrate e le relative mitigazioni in atto. Questo documento è rappresentato dal Threat Model, ed è importante che sia presente all'interno del proprio processo, in modo da poter tenere sotto controllo l'andamento della sicurezza del proprio software, ad esempio verificando che con il passare del tempo il numero delle vulnerabilità diminuisca.

Concludiamo questa introduzione sull'attività di Threat Modeling, facendo notare che questa è una di quelle attività che dovrebbe essere portata avanti di pari passo con la fase di design del software. Non deve essere un'attività per la quale basta spuntare una casella per dire che è stata fatta. La creazione del Threat Model è di tipo incrementale, e man mano che l'architettura e il design dell'applicazione diventano più nitidi, si riesce ad intravedere i primi punti in cui è possibile minacciare la sicurezza del software sviluppato. Ulteriori informazioni sul threat modeling possono essere trovate in [33,23], inoltre Microsoft mette a disposizione un tool per la stesura del Threat Model, per il download si veda [25].

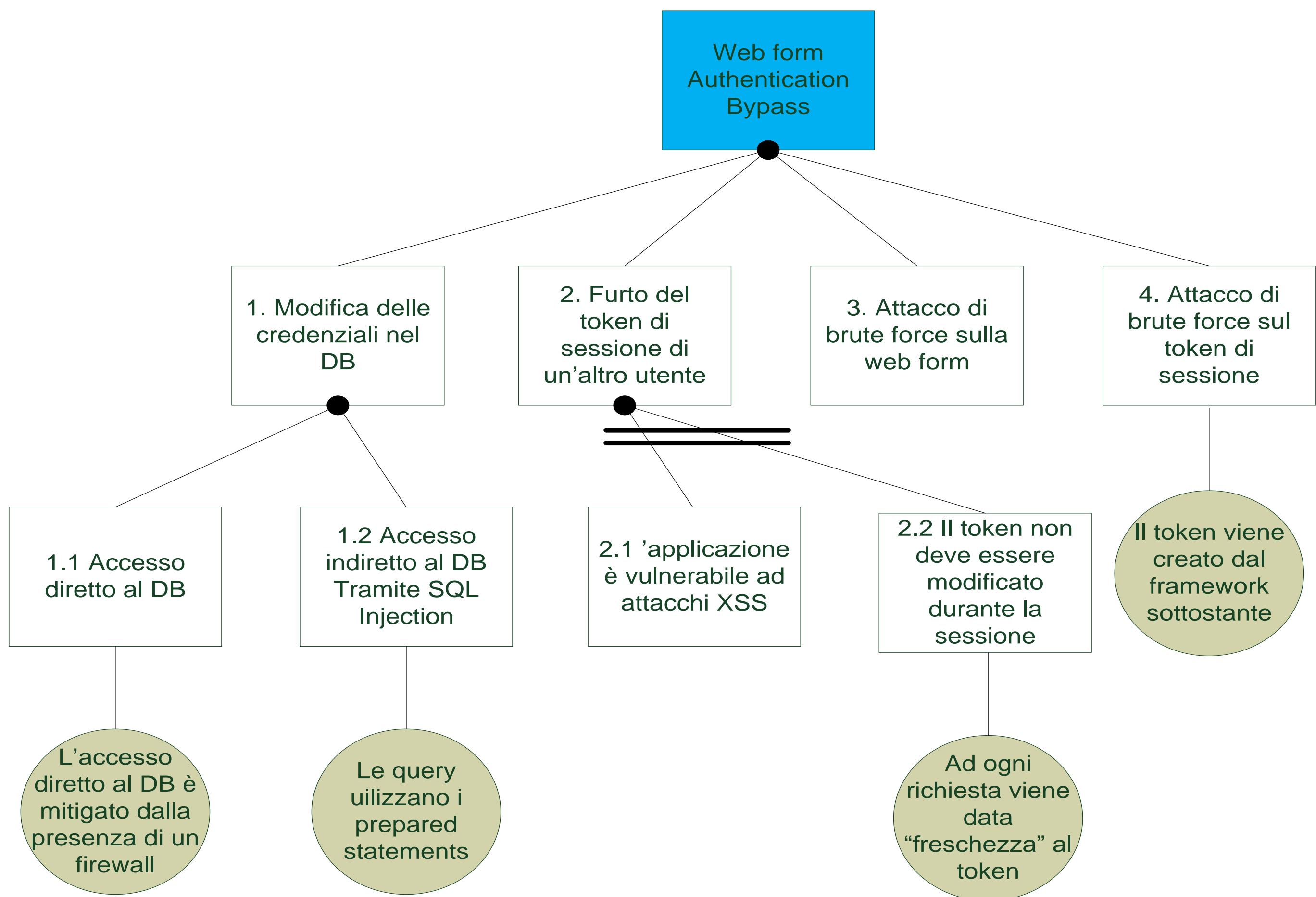

Figura 3. Esempio di attack tree per l'analisi della minaccia "Web Fom Authentication Bypass" in un'applicazione web

3.2.4 Integrazione del Threat Modeling nel SDLC

Il Threat Modeling è un'attività che come già detto viene svolta durante la fase di design. La sua integrazione è semplice, basta inserire un'ulteriore fase in parallelo a quella di design all'interno del proprio processo. L'unica attenzione da prendere è quella di evitare di completare il design e poi cominciare l'attività di threat modeling, perché in questo modo si perderebbero i benefici iniziali introdotti, e ci si ritroverebbe a modificare e nei casi peggiori a stravolgere il design dell'applicazione nel caso in cui risultasse altamente insicuro.

3.3 Implementazione e Testing

In questo paragrafo raggrupperemo tutte le informazioni relative alle fasi di

- Pianificazione test
- Implementazione
- Testing e risultati

in queste fasi viene fatto uso di tools per automatizzare i processi di testing e analisi. Cominciamo a parlare innanzitutto della parte di Implementazione, per poi passare a parlare della parte di testing. Per quel che riguarda la loro integrazione all'interno del processo di sviluppo, si eviterà di dedicare un paragrafo a parte per ogni fase, dato che si tratta semplicemente di modificare attività già esistenti (o per lo meno si spera che in un ciclo di sviluppo software sia presente l'attività di testing).

3.3.1 Implementazione sicura

Nella fase di implementazione, è importante avere degli standards ben definiti per la scrittura del codice. All'interno dell'azienda dovrebbe essere presente un documento che specifica quali sono le pratiche di sicurezza da seguire. Generalmente in tali documenti viene descritto quali sono le librerie da usare, quali funzioni usare, e ovviamente quali funzioni non usare. Per rafforzare tali politiche si fa spesso uso di alcuni "stratagemmi" all'interno del codice, in modo che quando uno sviluppatore cerca di utilizzare delle funzioni definite *unsafe* il compilatore rileva un errore di sintassi.

Sempre nella fase di implementazione trovano posto i tools di analisi statica del codice sorgente. Indubbiamente il loro utilizzo è da considerarsi fondamentale, perché permettono

di rilevare banali errori di sicurezza durante la scrittura del codice. Questi tools sono in genere integrati all'interno del proprio IDE per una maggiore usabilità. Va da sé, che l'utilizzo dei tools di analisi statica non deve comunque sostituire l'attività manuale di security code review. Questa attività è sicuramente tra le più proficue per quel che riguarda la rilevazione degli errori, ed è brevemente descritta nel prossimo paragrafo. Per quel che riguarda la scrittura di codice sicuro per ulteriori informazioni si faccia riferimento a [14].

3.3.1.1 Processo di security code review

È importante includere nel proprio processo di sviluppo delle sessioni di security code review (anche detto software security assessment). Per eseguire un security code review, è prima necessario comprendere in che modo funziona l'applicazione. Se il code review è fatto all'interno dell'azienda questa fase dovrebbe essere data per scontata, se invece il code review è demandato ad un ente esterno, è necessario come prima cosa disporre di un diagramma di alto livello del funzionamento dell'applicazione, documenti come il Threat Model in questa fase sono di utile aiuto, perché contengono già parte dei diagrammi che saranno di interesse a chi dovrà effettuare il code review.

Una volta compresa a grandi linee la logica applicativa, vi sono varie strategie per eseguire una code review del software, le due principali sono

- *bottom up:* in questo tipo di approccio, si fa meno uso della documentazione riguardo il funzionamento dell'applicazione e si procede con l'identificazione delle vulnerabilità di più basso livello. Detto in altre parole nella metodologia bottom up si comincia dall'implementazione e tramite la sua analisi si procede con lo scovare le vulnerabilità di immediata identificazione. Con il progredire dell'analisi si costruisce un modello dell'applicazione fino a riuscire, da tale modello, ad identificare le vulnerabilità più complesse che spaziano attraverso svariati file sorgente. Il suo pregio è quello di riuscire col tempo ad ottenere una comprensione dell'applicazione altamente accurata, con lo svantaggio che è un processo lento, e che a volte porta ad esaminare porzioni del programma che non intaccano direttamente la sua sicurezza (ad esempio le routine di creazione della GUI).
- *top down:* nell'approccio top down invece si comincia già disponendo di una struttura dell'applicazione, anche se non del tutto fedele. Da tale struttura si identificano quali sono i punti critici da analizzare per primi. In questo approccio si identificano per prime le vulnerabilità riguardanti il design, per poi procedere con quelle logiche, fino a scendere a quelle di più basso livello. Questo approccio dà buoni risultati se esiste un modello fedele dell'applicazione, in caso contrario si rischia di trascurare alcune parti che invece andrebbero esaminate.

Prima di concludere il paragrafo accenniamo alla modalità con cui andrebbe effettuato un code review. Esso è in genere compiuto in sessioni brevi di circa 4-5 ore al massimo (di più si comincerebbe a perdere lucidità), con pausa ogni 1-2 ore. È bene che sia disponibile anche lo sviluppatore del codice che si sta esaminando, per eventuali chiarimenti. Inoltre è importante non soffermarsi sulla risoluzione del problema, ma bisogna limitarsi alla loro esclusiva identificazione. Le sessioni di code review vengono in genere svolte in gruppo, e risultano più proficue se i membri presentano abilità di analisi diverse, in questo modo ogni partecipante si può concentrare sull'identificazione delle vulnerabilità per le quali possiede un'approfondita conoscenza.

Il processo di code review è molto complesso e in questa sede è stato solamente accennato, per ulteriori informazioni si rimanda a ciò che può essere considerato il testo per antonomasia sul security code review ([7]).

3.3.2 Security Testing

Giungiamo infine a una delle due attività che sono probabilmente le più diffuse per quanto concerne l'analisi della sicurezza del software. A differenza del security code review, la quale è un'attività di analisi, ovvero non è necessaria l'esecuzione del software, il testing è un'attività in cui si lancia in esecuzione il software e si effettuano alcuni test di sicurezza. Con security testing non si intende soltanto l'attività di penetration test (trattata nel prossimo paragrafo), ma anche attività di testing che non riguardano l'intera applicazione. I security tests trovano utilizzo principalmente durante il processo di sviluppo software e più raramente in uno scenario post sviluppo. La loro applicazione è del tutto simile ai normali test funzionali, con la sola differenza che si punta a individuare errori riguardanti la sicurezza. Cosi come per i normali test funzionali anche per i security tests è necessario prima costruire l'ambiente necessario al testing, per cui scrivere gli eventuali driver e stubs, tale attività prende il nome di *scaffolding* ([12]). Citiamo infine che per sfruttare al meglio i security tests, si può eseguire una procedura di testing guidata dal threat model. In questo modo si sa già cosa andare a testare e in che modo.

3.3.3 Penetration Testing

Il penetration testing è sicuramente l'attività di testing più diffusa. Trova posto generalmente in uno scenario post sviluppo (nonostante sia parte fondamentale all'interno del SDLC, si veda a tal proposito [39]), e viene effettuato generalmente da un ente esterno. È credenza comune che con attività di penetration testing si possa incrementare sostanzialmente la sicurezza della propria applicazione. Purtroppo ciò non è vero nella maggior parte dei casi. L'attività di penetration testing trova il massimo utilizzo nel verificare la sicurezza dell'installazione e della configurazione del sof-

tware. Ciò perché le vulnerabilità identificate da attività di penetration testing tradizionali tendono a scoprire "semplici bug" che possono generalmente essere evitati eseguendo una delle tante attività fino ad ora menzionate. Non bisogna comunque cadere nell'errore di pensare che attività di testing in particolare di tipo black box (come appunto il penetration testing) siano inutili, a supporto di ciò si consideri il bug integer overflow in ssh documentato in [38], il quale è stato identificato tramite un semplicissimo test di tipo black box (ssh –l long_user_name) ma che presenta un'analisi alquanto complessa e difficile da identificare tramite attività di code review. Terminiamo qui la discussione sull'attività di testing e penetration testing, per ulteriori informazioni si faccia riferimento a [19,28].

3.3 Feedback esterno

Cominciamo il paragrafo con un'affermazione ben nota agli esperti di sicurezza informatica: "la sicurezza assoluta non esiste". Nonostante si applichino tutte le attività qui esposte, non si può comunque escludere la presenza di vulnerabilità nel proprio software. Per questo è necessario predisporre di un'attività che abbia il compito di ricevere eventuali riscontri di vulnerabilità, e di mettere in moto un processo che miri a:

- identificare la vulnerabilità
- identificare la causa
- verificare se la vulnerabilità è presente in altri punti del codice
- porvi un rimedio
- aggiornare i propri tools di analisi e i propri documenti in modo che tale vulnerabilità non venga più introdotta nei successivi prodotti

Un tale processo, può far risparmiare all'azienda molto tempo e risorse.

4 DIAGRAMMA DELLE ATTIVITÀ

Sono state presentate svariate attività, tutte con il solo scopo di incrementare la sicurezza del software sviluppato. In **Figura 4** è presentato un diagramma di attività in cui si è evidenziato quali sono i collegamenti tra le varie attività e dove dovrebbero essere inserite all'interno del ciclo di sviluppo software.

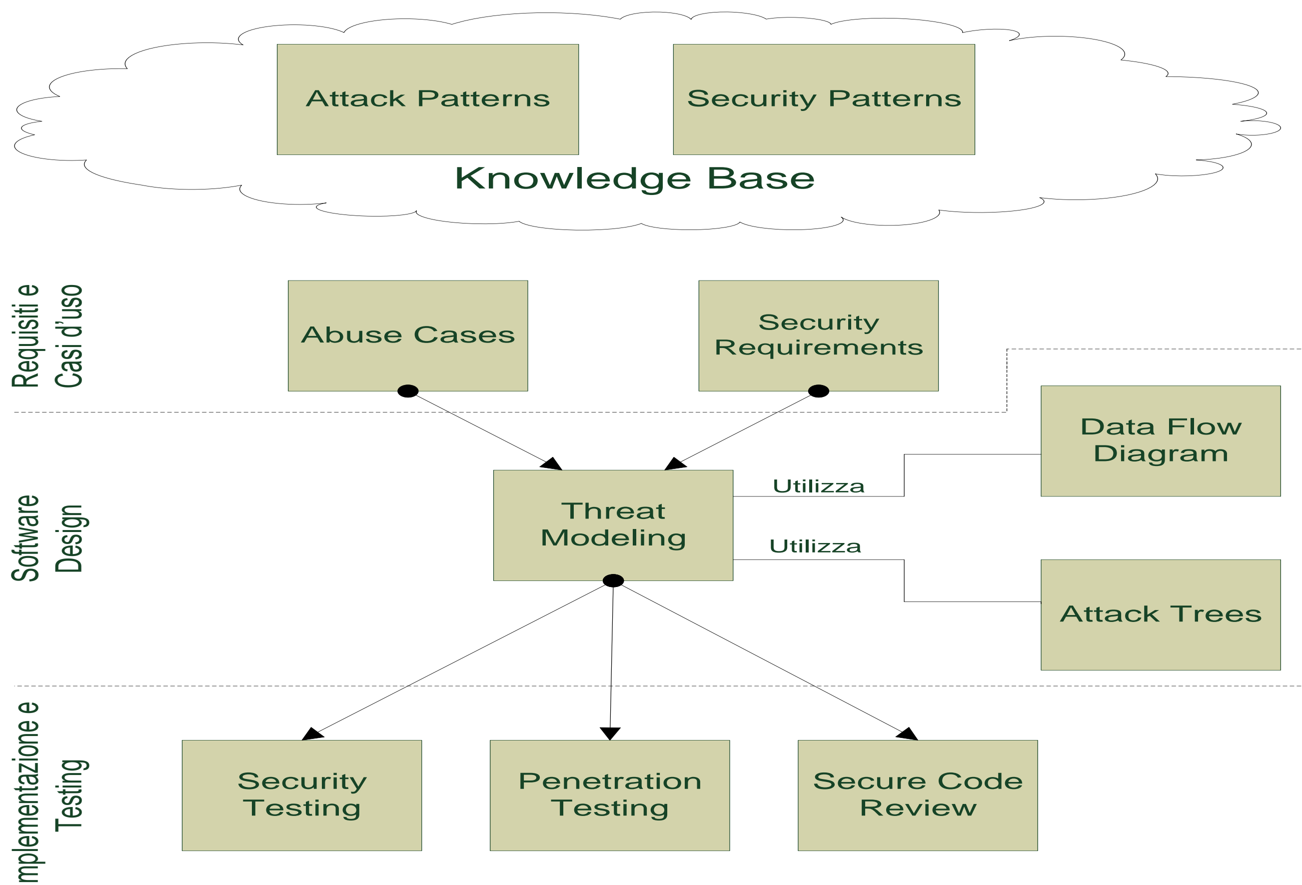

Figura 4. Diagramma delle attività di sicurezza

Trovandosi a dover integrare per la prima volta tali attività all'interno del proprio ciclo di sviluppo software, è consigliabile seguire un approccio incrementale, in questo modo si evita di sconvolgere troppo il proprio processo, e gli si permette di adattarsi con il tempo.

Un dubbio spontaneo potrebbe essere, quali attività dovrebbero essere implementate per prime? Una possibile risposta a tale domanda è rappresentata dalla seguente classifica:

1. Secure Code Review, compreso l'utilizzo dei software di analisi statica del codice sorgente
2. Threat Modeling, compreso di Data Flow Diagram e Attack Trees
3. Penetration Testing, compreso l'utilizzo dei relativi software
4. Security Testing, compreso l'utilizzo dei relativi software
5. Attack Patterns e Security Patterns
6. Abuse Cases
7. Security Requirements

5 CONCLUSIONI E SVILUPPI FUTURI

Negli ultimi anni la sicurezza si è spostata sempre di più dal livello infrastrutturale al livello software. In particolar modo sempre più spesso si richiede, oltre all'immancabile penetration test, anche servizi di security code review. Allo stato attuale è comunque ancora radicata l'idea di occuparsi del fattore sicurezza in uno scenario post-sviluppo. È molto probabile che non appena le aziende produttrici di software si renderanno conto che possono risparmiare prevenendo l'introduzione di vulnerabilità di sicurezza già dalle prime fasi del ciclo di sviluppo, l'attenzione andrà a spostarsi sempre più verso le fasi iniziali del processo di sviluppo. Oltre a questo spostamento, si cominciano ad intravedere all'orizzonte i primi modelli per stabilire quanto un software sia sicuro ([22]), e a fornire certificazioni sulla sicurezza delle applicazioni che non abbiano un metodo di applicazione troppo tedioso ([23]).

6 RIFERIMENTI

[1] Mike Andrews, James A. Whittaker. *How to break Web Software*. 2006.

[2] Sean Barnum, Amit Sethi. Attack Pattern Generation. Disponibile a <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/586.html?branch=1&language=1>

[3] Sean Barnum, Amit Sethi. Attack Pattern Usage. Disponibile a <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/588.html?branch=1&language=1>

[4] Sean Barnum, Amit Sethi. Introduction to Attack Pattern. Disponibile a <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/585.html?branch=1&language=1>

[5] CAPEC. Common Attack Pattern Enumeration and Classification. Disponibile a <http://capec.mitre.org/>

[6] Common Criteria Portal. Common Criteria. Disponibile a <http://www.commoncriteriaportal.org/>

[7] Mark Dowd, John McDonald, Justin Schuh. *The art of Software Security Assessment*. 2006.

[8] Federal Aviation Administration. *integrated Capability Maturity Model.* Disponibile a <http://www.faa.gov/about/office_org/headquarters_offices/aio/business_value/icmm/>

[9] Dana Epp. *Secure Software Programming: DREAD is Dead.* Disponibile a <http://silverstr.ufies.org/blog/archives/000875.html>

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[11] Michael Gegick, Laurie Williams. Matching Attack Patterns To Security Vulnerabilities in Software-Intensive System Designs in *ICSE-SESS 05*, 2005

[12] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli. *Fundamentals Of Software Engineering – Second Edition*. 2003.

[13] Mark G. Graff, Kenneth R. van Wyk. *Secure Coding, Principles & Practices*. 2003.

[14] Johan Grégoire, Koen Buyens, Bart De Win, Riccardo Scandariato, Wouter Joosen. *On the Secure Software Development Process: CLASP and SDL Compared.* Disponibile a <http://homes.dico.unimi.it/~monga/lib/sess07/28300046.pdf >

[15] Greg Hoglund, Gary McGraw. *Exploting Software, hot to break code*. 2004.

[16] Michael Howard, Steve Lipner. *The Security Development Lifecycle*. 2006.

[17] Michael Howard, David LeBlanc. *Writing Secure Code – Second Edition*. 2003.

[18] International Systems Security Engineering Association. Systems Security Engineering – Capability Maturity Model. Disponibile a <http://www.sse-cmm.org/model/model.asp>

[19] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, Riley Hassell. *The Shellcoder's Handbook*. 2004.

[20] Steve Lipner, Michael Howard. *The trustworthy Computing Security Development Lifecycle*. Disponibile a <http://msdn2.microsoft.com/en-us/library/ms995349.aspx>

[21] Steve McConnel. *Code Complete – Second Edition*. 2004

[22] Gary McGraw. *Software Security – Building Security In*. 2006.

[23] J.D. Meier, Alex Mackman, Blaine Wastell. *Threat Modeling Web Application*. Disponibile a

<http://msdn2.microsoft.com/it-it/library/ms978516.aspx>

[24] Microsoft Threat Analysis & Modeling v2.1.2. Disponibile per il download a <http://www.microsoft.com/downloads/details.aspx?familyid=59888078-9daf-4e96-b7d1-944703479451&displaylang=en>

[25] Andrew P. Moore, Robert J. Ellison, Richard C. Linger. *Attack Modeling for Information Security and Survivability*. Disponibile a <http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn001.pdf >

[26] OWASP. Application Security Metrics Project. Disponibile a <http://www.owasp.org/index.php/Category:OWASP_Application_Security_Metrics_Project>

[27] OWASP. Comprehensive, Lightweight Application Security Process. Disponibile a <http://www.owasp.org/index.php/Category:OWASP_CLASP_Project>

[28] OWASP. The OWASP Testing Guide. Disponibile a <http://www.owasp.org/index.php/Category:OWASP_Testing_Project>

[29] OWASP. The OWASP Web Security Certification Framework. Disponibile a <http://www.owasp.org/index.php/SpoC_007_-_The_OWASP_Web_Security_Certification_Framework>

[30] Antonio Parata. Valutazione del rischio tramite la logica Fuzzy, *Smau E-Accademy 2007*.

[31] Bruce Schneier. *Secrets and Lies – Digital Security in a Networked World*. 2000.

[32] Software Engineering Institute. Capability Maturity Model. Disponibile a <http://www.sei.cmu.edu/cmm/>

[33] Software Engineering Institute. Capability Maturity Model Integration. Disponibile a <http://www.sei.cmu.edu/cmmi/>

[34] Christopher Steel, Ramesh Nagappan, Ray Lai. *Core Security Patterns – Best Practices and strategies for J2EE, Web Services, and Identity Management*. 2006.

[35] Frank Swiderski, Window Snyder. *Threat Modeling*. 2004.

[36] James A. Whittaker, Herbert H. Thompson. *How to break Software Security*. 2003.

[37] Kenneth R. van Wyk. *Adapting Penetration Testing for Software Development Puposes*. Disponibile a <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration/655.html>

[38] Michal Zalewski. *SSH CRC32 attack detection code contains remote integer overflow*. 2001.