| | |
|---|---|
| Asset | Vengono elencati gli asset che potrebbero essere di interesse per un potenziale avversario. Questi possono essere di tipo fisico o logico. Possibili esempi di asset sono: le credenziali di accesso, i dettagli di una carta di credito, le chiavi di crittografia o i dettagli dell'utente. |
| Punti di Ingresso/Uscita | I punti di ingresso/uscita sono quei punti del sistema dove i dati entrano o escono. Questi vengono talvolta indicati come "punti di attacco" poiché soggetti ad uno o più potenziali attacchi. Ciascuno di questi, corrisponde ad una parte del sistema per la quale dovrebbero essere implementate opportune misure di sicurezza. |
| Livelli di Fiducia o di Trust | Un livello di trust viene utilizzato per definire i privilegi che un'entità esterna deve avere per accedere al sistema. Questi possono essere classificati in base ai privilegi assegnati o alle credenziali fornite e fanno riferimento a punti di ingresso/uscita di risorse protette. Possibili esempi sono l'utente anonimo del sistema, l'utente autenticato del sistema e l'amministratore. I livelli di trust vengono applicati ad ogni punto di entrata/uscita. |

## 5.2 Secure by Design

Le applicazioni la cui architettura non viene valutata dal punto di vista della sicurezza sono applicazioni 'fragili'. L'analisi, fatta in concerto sia sull'architettura dell'applicazione che sulle interazioni, accresce la visibilità dei team IT circa gli ambiti della sicurezza e del risk management. Gli architetti delle applicazioni software sono responsabili della realizzazione del progetto per garantire l'adeguata copertura ai rischi derivanti sia da un uso standard del sistema, sia da un attacco dannoso.
Nella fase di design di una nuova applicazione o di re-ingegnerizzazione di un'applicazione esistente, per ciascuna caratteristica funzionale, è necessario analizzare aspetti aggiuntivi quali:

- Gli aspetti di sicurezza nel processo riguardante tale funzione sono stati massimizzati?
- Premeditatamente, come si potrebbe abusare di tale funzione?
- La funzione deve essere attivata per impostazione predefinita? In caso affermativo, esistono limiti o alternative che potrebbero contribuire a ridurre il rischio derivante da tale funzione?

Andrew van der Stock chiama il suddetto processo "Thinking Evil", e raccomanda di mettersi nei panni dell'attaccante e di pensare a tutte le possibili alternative per abusare di ogni singola funzione, considerando i tre pillar fondamentali (Riservatezza, Integrità, Disponibilità) e utilizzando a sua volta il modello STRIDE.
Utilizzando il modello di rischio di minaccia STRIDE/DREAD discusso nel presente documento, è possibile intraprendere la giusta strada per adottare formalmente un'architettura sicura nello sviluppo applicativo del software.
Il concetto di sicurezza nell'architettura inizia ad esistere nel momento in cui vengono modellati i requisiti di business e prosegue fino a quando l'ultima istanza dell'applicazione viene dismessa.

### 5.2.1 Principi base del secure design

"Security by design" è un concetto base dell'ingegneria del software e definisce un software progettato espressamente per essere sicuro (anticipando e minimizzando a priori gli impatti delle vulnerabilità che potrà manifestare in produzione) capace di garantire i seguenti caposaldi della sicurezza dell'informazione:

- **R**iservatezza: consentire l'accesso solo ai dati per i quali l'utente è autorizzato.
- **I**ntegrità: garantire che i dati non vengano manomessi o alterati da utenti non autorizzati.

- **D**isponibilità: garantire che i sistemi e i dati siano sempre disponibili agli utenti autorizzati quando questi ne hanno bisogno.

I principi base del Secure Design possono essere sintetizzati come segue:

1) Ridurre al minimo la superficie di attacco.
   Ogni funzione aggiunta a un'applicazione immette una certa dose di rischio nell'applicazione. L'obiettivo di uno sviluppo sicuro è quello di ridurre il rischio complessivo riducendo la superficie di attacco. Ad esempio, un'applicazione web implementa un Help on line con una funzione di ricerca. La funzione di ricerca può essere vulnerabile agli attacchi di SQL injection. Se l'accesso alla funzione di Help viene limitata ai soli utenti autorizzati, la probabilità di attacco è certamente ridotta. Se la funzione di ricerca relativa all'Help è stata implementata attraverso routine di convalida dei dati centralizzate, la capacità di eseguire l'iniezione SQL viene drasticamente ridotta. Tuttavia, se la funzione di Help è stata riscritta per eliminare la funzione di ricerca (ad esempio attraverso un miglioramento dell'interfaccia utente), ciò elimina quasi totalmente la superficie di attacco, anche se la funzione di Help in generale rimane disponibile su Internet.

2) Stabilire valori predefiniti sicuri.
   Esistono diversi modi per offrire agli utenti un'esperienza "out of the box". Tuttavia, per impostazione predefinita, tale esperienza dovrebbe essere sicura e dovrebbe spettare all'utente ridurre il proprio livello di sicurezza, se consentito. Ad esempio, la scadenza e la complessità della password dovrebbero essere abilitate per impostazione predefinita. Gli utenti potrebbero comunque essere autorizzati a disattivare tali funzioni per semplificare l'uso dell'applicazione di conseguenza incrementando il loro livello di rischio.

3) Adottare il principio del minimo privilegio.
   Il principio del "minimo privilegio" suggerisce che agli account venga dato il livello minimo di privilegio richiesto necessario per poter eseguire i relativi processi di business. Ciò include i diritti dell'utente, i permessi di accesso alle risorse come la limitazione nell'uso della quota di CPU, i permessi di accesso alla memoria, alla rete e al file system. Ad esempio, se un server middleware richiede solo l'accesso alla rete, l'accesso in lettura a una tabella di database e la possibilità di scrivere su un log, ciò definisce tutte le autorizzazioni che dovrebbero essere concesse. In nessun caso al middleware devono essere concessi privilegi amministrativi.

4) Adottare il principio di difesa a più livelli o in profondità.
   Il principio della difesa a più livelli anche detta "Defense in Depth", suggerisce che laddove un controllo può risultare adeguato, più controlli che affrontano i rischi in modi diversi possono rappresentare l'approccio migliore. Quando tali controlli vengono eseguiti in profondità, questi possono rendere le più gravi vulnerabilità presenti nel sistema incredibilmente difficili da sfruttare e quindi improbabili. In termini di codifica sicura, lo si può tradurre in una forma di validazione basata su più livelli, in controlli di auditing centralizzati, richiedendo che tutti gli utenti vengano autenticati su tutte le pagine dell'applicazione. Ad esempio, è improbabile che un'interfaccia amministrativa che presenta delle difettosità sia vulnerabile a possibili attacchi anonimi se l'accesso alla rete di gestione dell'ambiente di produzione viene opportunamente bloccata, viene controllata l'autorizzazione amministrativa dell'utente e vengono registrati su log tutti gli accessi.

5) Gestire gli errori in modo sicuro.
   A causa di diversi motivi, le applicazioni software falliscono durante la propria esecuzione. La modalità in cui queste falliscono determina il fatto che siano sicure o meno. La gestione sicura degli errori è un aspetto fondamentale per le applicazioni sicure e resilienti. Esistono due tipi principali di errori che richiedono una particolare attenzione:
   - le eccezioni che si verificano durante l'elaborazione di un controllo di sicurezza,

- le eccezioni generate dal codice che non sono "rilevanti per la sicurezza".

È importante che queste due distinte tipologie di eccezioni non permettano un comportamento che una contromisura software normalmente non consentirebbe. In termini di sviluppo, è opportuno considerare il fatto che un meccanismo di sicurezza possa dare seguito a tre distinti possibili risultati:

- consentire l'operazione;
- bloccare l'operazione;
- sollevare un'eccezione.

In generale, il meccanismo di sicurezza dovrebbe essere progettato in modo tale che un eventuale fallimento segua lo stesso percorso di esecuzione implementato per il blocco dell'operazione. Ad esempio, metodi di sicurezza come "isAuthorized" o "isAuthenticated" dovrebbero restituire tutti "false" se è stata sollevata un'eccezione durante l'elaborazione. Se gli stessi controlli di sicurezza prevedono delle eccezioni, è opportuno che ci sia chiarezza su cosa significhi esattamente ciascuna condizione di errore.

6) Non affidarsi ai servizi di terze parti o a risorse esterne.
Molte organizzazioni utilizzano le capacità di elaborazione di terzi parti, che molto spesso adottano politiche e postura di sicurezza significativamente diverse. È altamente improbabile che si possa influenzare o controllare tali parti, siano esse utenti privati, grandi fornitori o partner. Pertanto, è opportuno non affidarsi implicitamente a quei servizi/sistemi che vengono gestiti esternamente. Tutti i servizi/sistemi esterni devono essere trattati allo stesso modo.

7) Separare i ruoli.
Un controllo antifrode fondamentale è la separazione dei ruoli. Ad esempio, chi richiede un finanziamento non può essere l'unico garante, né dovrebbe ricevere direttamente il finanziamento. Ciò impedisce al soggetto di richiedere numerosi prestiti e di affermare poi di non averli mai avuti. Alcuni ruoli hanno livelli di fiducia diversi rispetto ai normali utenti. In particolare, gli amministratori sono diversi dagli utenti normali. In linea generale, gli amministratori non dovrebbero essere anche utenti dell'applicazione. Ad esempio, un amministratore dovrebbe essere in grado di accendere o spegnere il sistema, impostare le politiche sulle password, ma non dovrebbe essere in grado di accedere al frontend del sistema come utente super privilegiato, ed essere in grado di compiere un'azione per conto di altri utenti. La regola generale nella separazione dei ruoli è che l'entità che approva un'azione, l'entità che svolge un'azione e l'entità che controlla tale azione devono necessariamente essere separate. La separazione dei ruoli è un controllo importante sia nell'ambito finanziario che in quello informatico. L'obiettivo è quello di eliminare la possibilità che un singolo utente non compia e nasconda un'azione a lui non consentita. Separando i ruoli di autorizzazione, implementazione e monitoraggio, il tentativo di frode può essere portato a termine con successo solo attraverso la collusione delle parti. Lo sviluppo sicuro di una applicazione software dovrebbe sempre prevedere la separazione dei ruoli (ad esempio, implementando l'accesso basato sui ruoli) e tale concetto dovrebbe anche essere integrato nel ciclo di vita di sviluppo del dell'applicazione (ad esempio, limitando l'accesso degli sviluppatori alle librerie di produzione).

8) Segmentare l'infrastruttura di rete.
La segmentazione e la segregazione delle reti sono strategie altamente efficaci che un'organizzazione può attuare per limitare l'impatto dovuto ad un'intrusione di rete. Se implementate correttamente, tali strategie possono rendere significativamente più difficile per un avversario localizzare e accedere alle informazioni più sensibili dell'organizzazione aumentando la probabilità di rilevare tale attività in modo tempestivo. Storicamente, la segmentazione e la segregazione della rete veniva implementata attraverso l'impiego di un firewall perimetrale; oppure attraverso l'utilizzo di una coppia di firewall che delimitando una zona demilitarizzata fornendo un ambiente separato tra zone totalmente attendibili e non attendibili.

9) Ridurre i “single point of failure”.
   Un “single point of failure” è un componente critico del sistema che ha la capacità di arrestare le operazioni del sistema stesso in caso di guasto. L’adozione di tali elementi diviene inaccettabile in quei sistemi che richiedono affidabilità e disponibilità, come le applicazioni software, le reti o le supply chains. Dal punto di vista della sicurezza, la presenza di un “single point of failure” fa sì che nel momento in cui questo viene a mancare (a seguito di una qualsiasi causa) anche il sistema smette di funzionare inficiandone la disponibilità. Pertanto è opportuno progettare il software evitando per quanto possibile la scelta dei “single point of failure” definendo gli opportuni requisiti di ridondanza (load balancer, cluster, etc.).

10) Mantenere la semplicità nell’uso del sistema.
   È essenziale che la user interface sia progettata considerando la facilità d'uso, tale che gli utenti possano regolarmente e automaticamente applicare gli opportuni meccanismi di protezione. Inoltre, gli errori vengono ridotti al minimo nel momento in cui la percezione dell'utente rispetto ai propri obiettivi di protezione corrisponde ai meccanismi di sicurezza messi in campo e che l’utente stesso è obbligato ad usare. Tale principio stabilisce che la soluzione di sicurezza adottata deve essere comprensibile per gli utenti finali, sia nella sua fase di progettazione che durante il suo utilizzo (facilità di installazione, configurazione e usabilità), nascondendo la complessità introdotta dai meccanismi di sicurezza.

11) Non affidarsi ad una sicurezza basata sull’oblio.
   Questo principio afferma che l’adozione di tecniche basate sul concetto dell’occultazione nella risoluzione di problematiche di sicurezza non dovrebbero mai essere considerate affidabili. Se l’applicazione prevede come requisito di sicurezza che il suo URL di amministrazione venga nascosto in modo che possa essere sicuro, in realtà non lo è affatto. Diversamente, è più appropriato prevedere gli opportuni controlli di sicurezza al fine di mantenere sicura l'applicazione senza nasconderne le funzionalità di base o il codice sorgente. Implementare la sicurezza nascondendo le funzioni del sistema ritenute vulnerabili è da considerarsi un controllo di sicurezza debole, e quasi sempre fallisce quando questo è l'unico controllo messo in campo. Ciò non significa che mantenere la segretezza sia sbagliato, significa semplicemente che la sicurezza di quei sistemi ritenuti chiave non deve dipendere esclusivamente dal fatto che ne vengono tenuti nascosti i dettagli. Ad esempio, la sicurezza di un'applicazione non dovrebbe basarsi sulla segretezza del suo codice sorgente. Questa dovrebbe basarsi su numerosi altri fattori, tra cui politiche adeguate in materia di password, difesa in profondità, limitazione delle transazioni finanziarie o comunque critiche, architettura di rete solida, controlli antifrode e di audit. Un esempio pratico può essere considerato il sistema operativo Linux. Il codice sorgente di Linux è ampiamente disponibile, eppure, se adeguatamente protetto, può essere considerato un sistema operativo sicuro e robusto.

12) Mantenere i controlli di sicurezza semplici e chiari.
   Superficie d'attacco e semplicità vanno di pari passo. Alcune tendenze dell'ingegneria del software preferiscono approcci eccessivamente complessi alla semplicità e alla chiarezza del codice. Chi sviluppa dovrebbe evitare l'adozione di architetture complesse quando un approccio più chiaro e lineare sarebbe più rapido e semplice da proteggere. Quando si pensa alla sicurezza del software, la prima domanda che ci pone è: "In quanti modi diversi il nostro software potrebbe essere attaccato? Ovvero, quante "vie d'accesso" esistono nella nostra applicazione? E' un po' come chiedere "Quante porte e finestre ci sono in un edificio? Se l'edificio ha solo una porta esterna, è molto facile proteggere quella porta. Se diversamente, si dispone di mille porte, allora sarà quasi impossibile mantenere sicuro l'edificio, non importa quanto siano resistenti le porte o di quante guardie di sicurezza si dispone. E' importante dunque limitare le "vie d'accesso" al software ad un numero ragionevole, altrimenti questo non sarà mai sicuro. Ciò lo si ottiene rendendo l'intero sistema relativamente