ExploitFarm Project

Analisi di progetto e documentazione

Domingo Dirutigliano

Politecnico di Bari
Software Engineering

Contents

1 Introduzione **3**
- 1.1 Descrizione generale . . . 3
- 1.2 Obiettivi . . . 3
- 1.3 Perchè lo sviluppo di un nuovo attacker/submitter? . . . 3

2 Specifiche **3**
- 2.1 Composizione del progetto . . . 3
  - 2.1.1 Backend . . . 4
  - 2.1.2 Frontend . . . 5
  - 2.1.3 CLI (xfarm) . . . 5
- 2.2 Specifica dei requisiti (FURPS+) . . . 6
  - 2.2.1 Functionality . . . 6
    - 2.2.1.1 Esecuzione Attacchi . . . 6
    - 2.2.1.2 Submission delle flag . . . 6
    - 2.2.1.3 Gestione di attacchi distribuiti . . . 6
    - 2.2.1.4 Gestione Exploit . . . 6
  - 2.2.2 Usability . . . 7
    - 2.2.2.1 Setup . . . 7
    - 2.2.2.2 Analisi statistiche e visualizzazione . . . 7
  - 2.2.3 Reliability . . . 7
    - 2.2.3.1 Dinamicità delle configurazioni . . . 7
    - 2.2.3.2 Error handling e managment . . . 7
    - 2.2.3.3 Resilienza del sistema . . . 7
  - 2.2.4 Performance . . . 7
    - 2.2.4.1 Bilancio dei processi nell'esecuzione di un attacco . . . 7
  - 2.2.5 Supportability . . . 8
    - 2.2.5.1 Adattabilità . . . 8
    - 2.2.5.2 Portabilità e avvio . . . 8
  - 2.2.6 + . . . 8
    - 2.2.6.1 Versioning . . . 8
    - 2.2.6.2 Licenza . . . 8
    - 2.2.6.3 Use-Case diagram . . . 8
- 2.3 Specifiche Tecniche . . . 9
  - 2.3.1 Database . . . 9
  - 2.3.2 Redis . . . 9
  - 2.3.3 Gestione funzionalità backend . . . 9
    - 2.3.3.1 API HTTP . . . 10
    - 2.3.3.2 Stats processor . . . 10
    - 2.3.3.3 Submitter Process . . . 10
    - 2.3.3.4 SocketIo Process . . . 10
  - 2.3.4 Funzionalità frontend . . . 10
  - 2.3.5 Strutturazione del client (xfarm) . . . 11
  - 2.3.6 Autobilanciamento del carico sul client per l'esecuzione degli attacchi . . . 11
  - 2.3.7 Gestione e versioning degli exploit . . . 12
  - 2.3.8 Controllo distribuito degli attacchi condivisi . . . 12
    - 2.3.8.1 Definizione dei parametri disponibili e calcolati dal sistema . . . 13

2.3.8.2 Valutazione della potenza computazionale di ogni client 14
2.3.8.3 Gestione del tempo di esecuzione per ogni attacco 14
2.3.8.4 Algoritmo di distribuzione degli attacchi 14
2.3.8.5 Algoritmo di calcolo del timeout 15

3 Project Managment **16**
3.1 Gestione generale 16
3.2 Kanban (github) 16
3.3 Scheduling 17
3.4 COCOMO Analysis 17
3.4.1 Effort Multipliers 18
3.4.2 Scale Factors 18
3.4.3 Risultato Finale 18
3.4.4 Duration 19
3.5 Risk Managment and Analysis 19
3.5.1 Stima dell'effort e requisiti 19
3.5.2 Rilascio di versioni non totalmente funzionanti 19
3.5.3 Rischio di rendere la piattaforma complessa gestire 19
3.6 Release Managment 19
3.7 Modello di Business 20

4 Analisi SWOT **20**
4.1 S: Punti di forza 20
4.2 W: Punti di debolezza 20
4.3 O: Opportunità 20
4.4 T: Minaccie 20

5 Sviluppi Futuri **20**

1 Introduzione

1.1 Descrizione generale

"Exploitfarm" è un software completamente dedicato alle competizioni CTF Attack/Defence, che si occupa principalmente di gestire la fase di attacco e ti tutto quello che conseguentemente questa fase richiede per essere eseguita correttamente, al fine di semplificare e velocizzare gli attacchi.
In generale Exploitfarm si occupa di attaccare in parallelo una serie di team (attacker) e di raccogliere ed inviare seguendo i criteri e limitazioni indicate per la competizione che si sta svolgendo le flag al gameserver (submitter).

1.2 Obiettivi

- Setup e installazione facile, veloce, personalizzabile e facilmente automizzabile
- Gestione delle risorse per gli attacchi dinamica e reattiva
- Scrittura degli exploit e dei test su questi semplificata
- Interfaccia intuitiva con avvio e configurazione intuitiva e rapida
- Keep track of anything: accumula dati sugli attacchi e ne permette un'analisi veloce ed intuitiva
- Rende semplice la condivisione/collaborazione sugli attacchi e la loro esecuzione
- Leggero da eseguire su qualsiasi piattaforma
- Gestione distribuita degli attacchi
- Configurazione e modifiche dinamiche

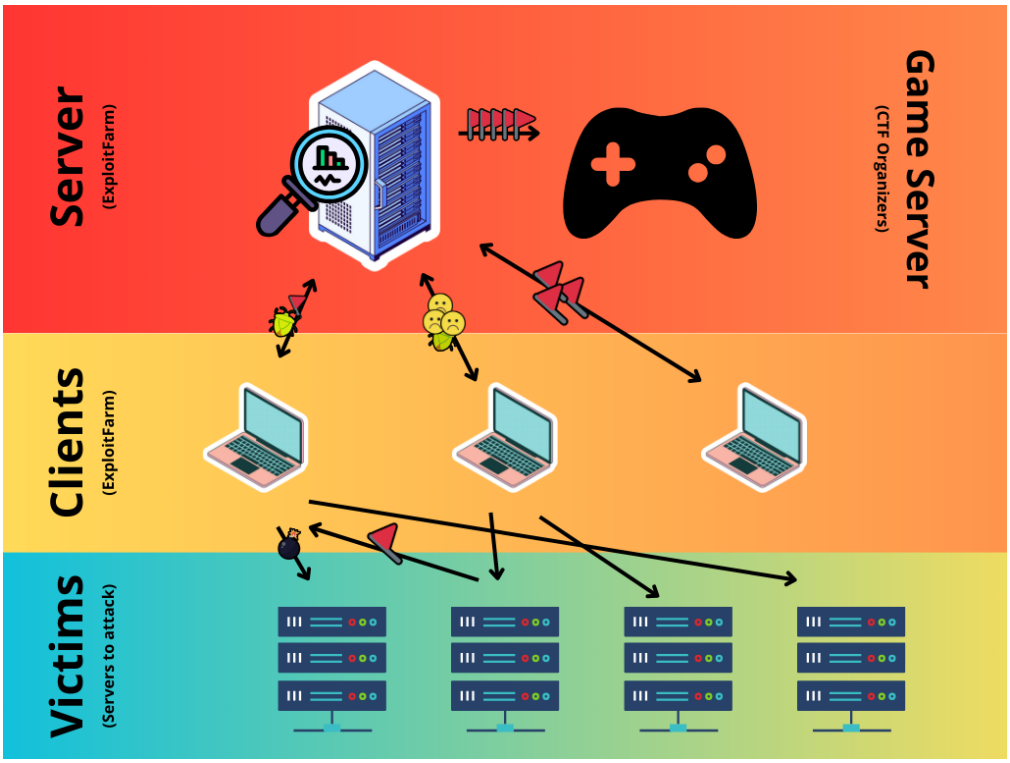
1.3 Perchè lo sviluppo di un nuovo attacker/submitter?

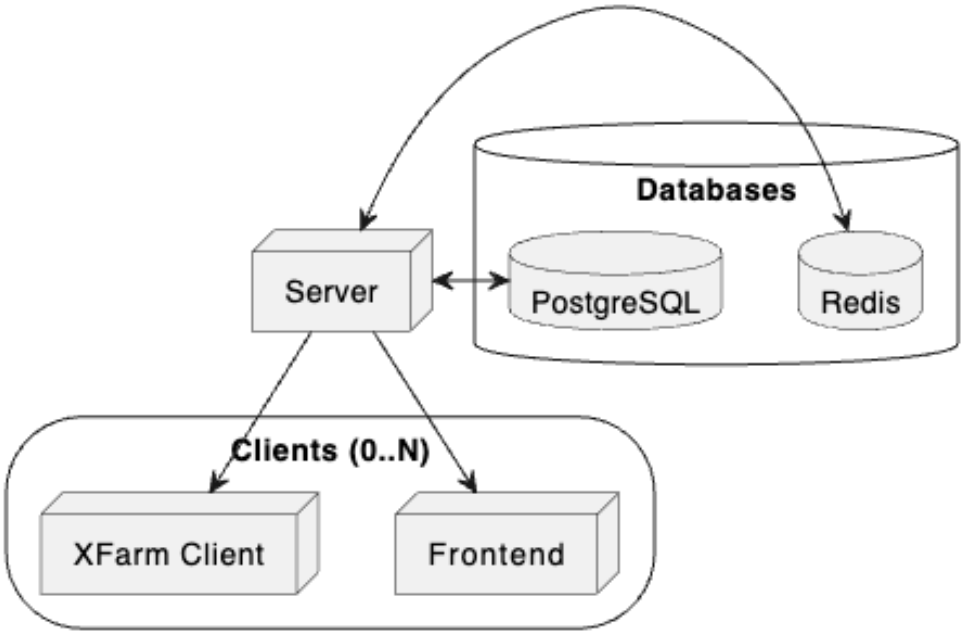
Gli attacker attualmente esistenti sono spesso incompleti, difficili da configurare e da completarne il setup, facilmente inclini ad errori che spesso comportano una perdita di tempo aggiuntiva, non hanno alcun tipo di gestione del carico supportato dalla macchina che esegue gli attacchi, non memorizza o espone alcun dato statistico sull'andamento degli attacchi ed infine non gestisce la condivisione degli exploit stessi.
Date le forti lacune presenti in software simili già esistenti, ho ritenuto opportuno la creazione di un'alternativa agli attacker attualmente esistenti.
NOTA: "ExploitFarm" è liberamente ispirato ad un altro attacker molto famoso chiamato DestructiveFarm, ma ne condivide a livello di codice unicamente delle piccole porzioni del suo client "start_xploit.py" a loro volta modificate ed adattate, in alcune parti riscritte e riprogettate completamente date le netti differenze di requisiti dei progetti.

2 Specifiche

2.1 Composizione del progetto

Il progetto è composto principalmente da un server centrale (nello specifico un web server) che coordinerà una serie di client sia frontend (web) che tramite una CLI. La parte di coordinamento, di submitting e gestione dei dati è affidata al server.





2.1.1 Backend

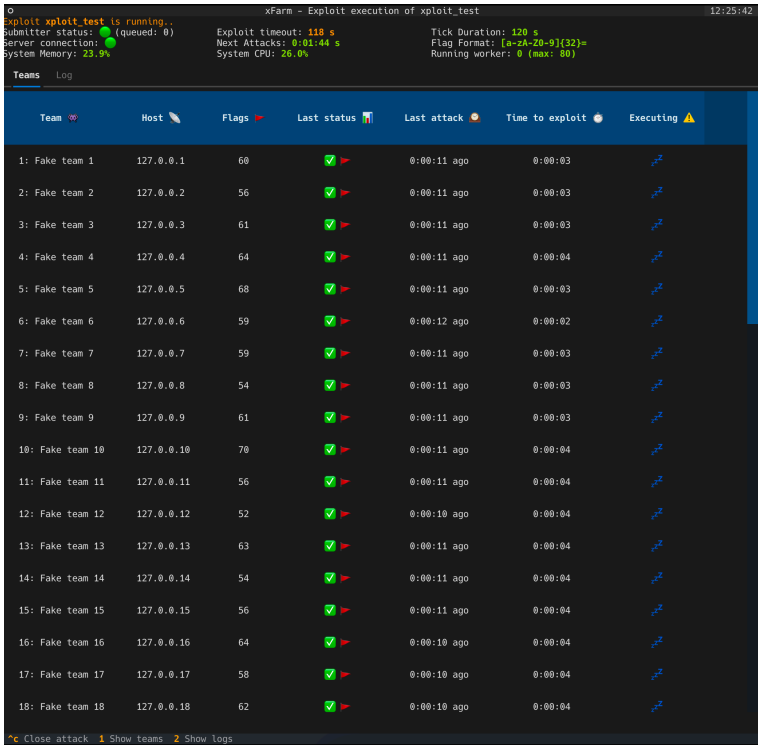
Il backend sarà il core del progetto poiché conterrà tutta la logica per il coordinamento dei vari client che invieranno il risultato degli attacchi, dovrà gestire i dati ed elaborarli al fine di renderli facilmente fruibili dai client, inoltre conterrà la logica e si occuperà della gestione del submitting delle flag al gameserver seguendo i requisiti indicati in fase di setup.

2.1.2 Frontend

La visualizzazione avanzata dello stato di ExploitFarm è invece affidata alla parte frontend del webserver che dovrà permettere un facile accesso ai dati presenti sul server, offrendoli tramite strumenti di analisi come grafici che devono essere mirati sulle esigenze decisionali che possono emergere durante una competizione Attack Defence. Inoltre dovrà segnalare e rendere facilmente e tempestivamente nota la presenza di eventuali errori di qualsiasi tipo sull'intera infrastruttura permettendone un'intervento quanto più immediato da parte del team.

2.1.3 CLI (xfarm)

Infine un'ultima parte fondamentale in tutto il progetto è il client che deve essere eseguito preferibilmente su macchine diverse da quella che offre il server, che si occupa dell'esecuzione stessa degli attacchi, della creazione del progetto dell'attacco, del monitoraggio (parziale) dell'attacco stesso. Anche il client stesso dovrà avere un'interfaccia in questo caso TUI intuitiva e veloce da utilizzare che deve rendere immediato e facile l'avvio dell'attacco e l'inserimento dei dati richiesti per l'esecuzione dell'attacco stesso.



2.2 Specifica dei requisiti (FURPS+)

2.2.1 Functionality

2.2.1.1 Esecuzione Attacchi

Il sistema deve essere in grado di eseguire un attacco scritto in qualsiasi linguaggio che rispetti determinati requisiti e prenda in input i dati necessari ad eseguire l'accatto, e li esegui in maniera coordinata monitorandone l'esecuzione e raccogliendo tutte le informazioni che potrebbero essere necessarie per analizzare la singola esecuzione dell'attacco. Gli attacchi dovranno essere inoltre coordinati per essere eseguiti seguendo un determinato criterio di scheduling configurato sul server

2.2.1.2 Submission delle flag

Il sistema deve essere in grado di raccogliere da tutti gli attacchi e filtrare le flag duplicate, e inviare queste seguendo i limiti imposti dal gameserver, tramite le configurazioni disponibili che ne devono permettere una gestione flessibile. Inoltre per allggerire il carico, il sistema deve anche automaticamente far scadere le flag che non soddisfano i requisiti di consegna per il gameserver.

2.2.1.3 Gestione di attacchi distribuiti

In caso di exploit con un carico computazionale e di memoria non indifferente come nel caso di alcuni attacchi criptografici o di attacchi che necessitano l'elaborazione di media, ExploitFarm deve disporre di un sistema per creare dei gruppi di client che eseguono lo stesso attacco ma i cui team da attaccare siano distribuiti in maniera bilanciata e basata sulla potenza computazionale dei client nel gruppo.

2.2.1.4 Gestione Exploit

I sorgenti degli attacchi devono poter essere facilmente condivisibili e avviabili anche da altri utenti.

Inoltre è richiesto anche tenere traccia delle versioni dell'attacco di modo da permetterne la segnalazione di vecchie versioni, aggiornamento di nuove versioni in maniera automatica in caso di attacchi condivisi (definiti in seguito), e analisi post-gara.

2.2.2 Usability

2.2.2.1 Setup

Il setup di ExploitFarm dalla sua installazione alla conclusione della sua configurazione deve essere di facile ed intuitivo utilizzo e di semplice finalizzazione. Al fine di perseguire questo obiettivo, la tecnologia per l'avvio del progetto è Docker, che ne permette facilmente di avviare un postgres e server redis dedicati e isolati. Inoltre è possibile avviare exploitfarm tramite un one-command, che avvierà il progetto da un repository pubblico con il container pre-buildato. Una volta avviato la configurazione iniziale dovrà essere compilabile ed eseguita sia tramite un'interfaccia web che tramite una automizzazione che è possibile scrivere tramite la libreria python associata al progetto.

2.2.2.2 Analisi statistiche e visualizzazione

Il sistema deve disporre di una interfaccia sempre aggiornata e facile da visionare per il monitoraggio delle flag in arrivo, degli attacchi eseguiti, che permetta la visualizzazione degli errori in caso di attacchi crashati o non funzionanti, di selezionare e filtrare i dati in base a dei parametri di ricerca e infine disporre di una serie di grafici che mostrino l'andamento degli attacchi durante la gara, di modo da permetterne una visione immediata.

2.2.3 Reliability

2.2.3.1 Dinamicità delle configurazioni

Tutte le configurazioni nella fase di setup devono essere modificabili e automaticamente aggiornate su tutti i client e in tutte le componenti del backend stesso, in modo da evitare interruzioni brusche degli attacchi, permettendo la continuità dell'esecuzione dei client e quindi dispendi di tempo nella riconfigurazione del sistema.

2.2.3.2 Error handling e managment

Se a causa di errori rilevabili c'è un errore nel sistema rilevabile, il sistema deve prontamente notificare su tutti i client l'errore/warning includendo tutte le informazioni utili a rilevarne l'origine, la motivazione e quindi velocizzando la risoluzione del problema.

2.2.3.3 Resilienza del sistema

Il sistema deve tentare in tutti i modi di impedire il riavvio ne tantomeno la chiusura degli exploit, e quindi anche a fronte di errori nei limiti di fattibilità continuare ad eseguire gli attacchi e consegnare le flag al gameserver. Ad esempio in caso il server vada offline gli attacker devono continuare ad accumulare flag fino a che il gameserver non tornerà online, e in caso di riavvio degli exploit mantenere la coda salvata per essere inviata successivamente. L'obiettivo deve essere non perdere flag ottenute e cercare di consegnarle al gameserver il prima possibile.

2.2.4 Performance

2.2.4.1 Bilancio dei processi nell'esecuzione di un attacco

Gli attacchi devono essere parallelizzati ed eseguiti tenendo conto delle risorse libere nel sistema: in caso il sistema arrivi in trashing o il processore sia saturo, il client deve ribilanciare automaticamente

il quantitativo di attacchi eseguiti al fine di non mandare il sistema in blocco. Cioè deve avvenire anche negli attacchi distribuiti.

2.2.5 Supportability

2.2.5.1 Adattabilità

Il sistema deve al limite di quanto possibile lasciare liberi gli utilizzatori di personalizzare ogni sua parte in modo dal renderlo adattabile a ogni CTF attack defence possibilmente senza modificare il source code del server o del client stesso. Inoltre gli attacchi devono poter essere scritti in qualsiasi linguaggio (nonostante tutta l'infrastruttura verti su python)

2.2.5.2 Portabilità e avvio

L'avvio del server deve essere rapido e semplice da configurare, inoltre deve facilmente adattarsi a più macchine e deve risultare semplice il passaggio di tutti i dati in un altro server, pertanto la migrazione deve essere quanto più intuitiva e rapida possibile.

2.2.6 +

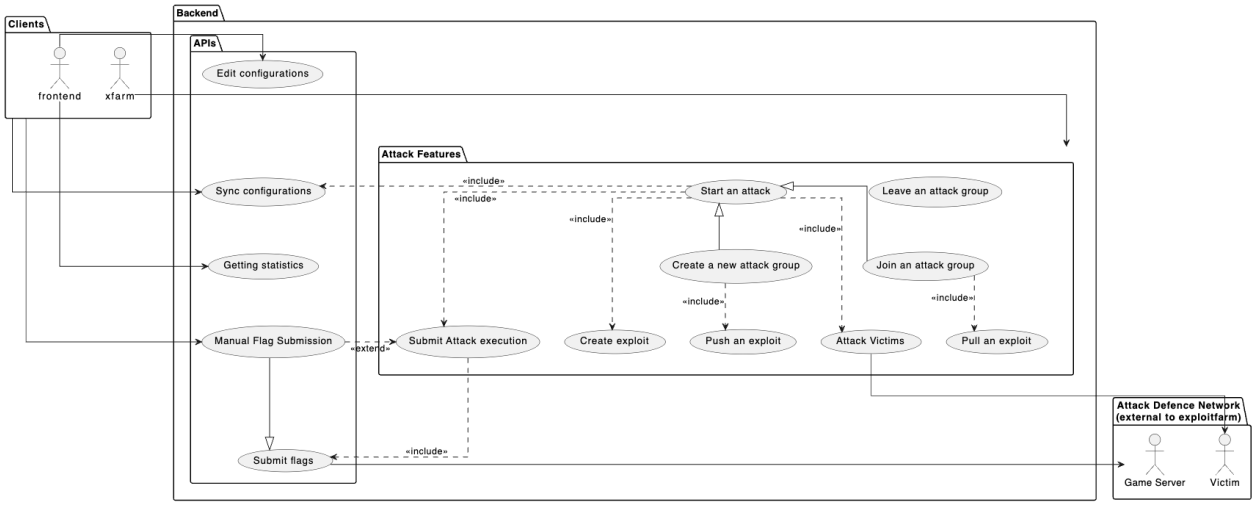
2.2.6.1 Versioning

Si utilizza un versioning a 3 cifre le quali sono la prima per la major release, che comporta dei cambiamenti stutturali al programma con possibile conseguente rottura di vecchi client, La seconda cifra indica la feature release che non influenza o provoca danni alle altre funzionalità ma è solo un'implementazione integrativa e non distruttiva, mentre l'ultimo numero indica la patch di quella versione in cui vengono risolti bug e crash inaspettati per le nuove funzionalità ma non solo. Viene fatta eccezione per le versioni con major 0. che sono consideranti ancora in sviluppo e non stabili, pertanto in queste versioni potrebbe verificarsi anche un cambio di API nelle feature release.

2.2.6.2 Licenza

Il progetto è open-source rilasciato con licenza GPL-3

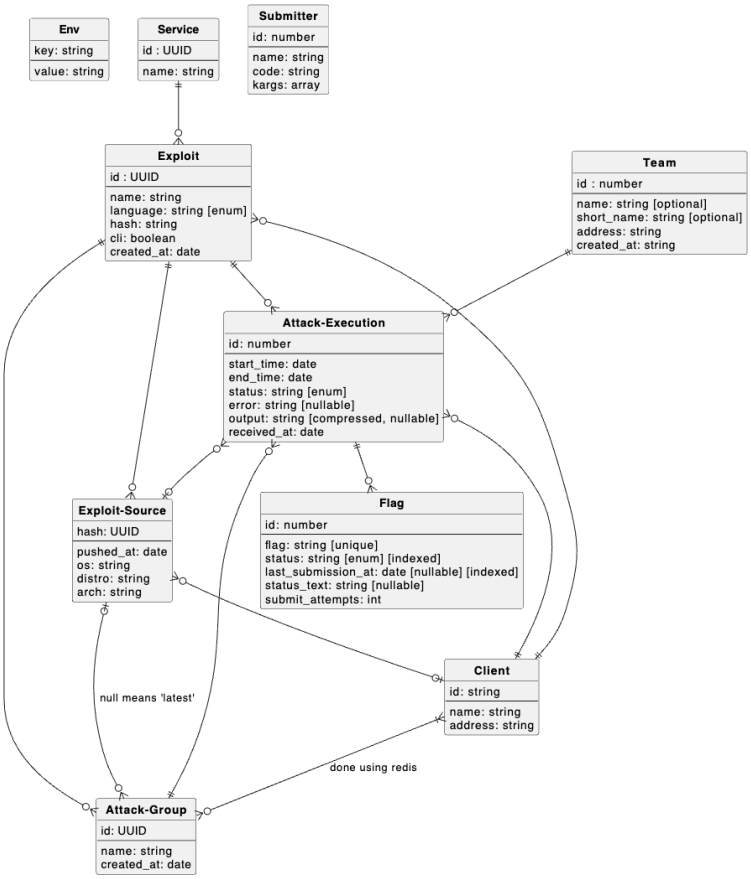
2.2.6.3 Use-Case diagram



2.3 Specifiche Tecniche

2.3.1 Database

Il database che gestisce i dati di ExploitFarm, data l'elevata mole di dati aspettata e la velocità di ricerca dei dati richiesta, è postgres e a seguito della stesura dell'analisi dei requisiti è stato progettato seguendo il seguente schema E/R



Alcune note aggiuntive:

- Env è una tabella che continene alcuni parametri di configurazione di ExploitFarm
- Submitter è una collezione di script python candidati come submitter per il sistema, essendo slegato dalla logica di attacco non ha alcuna relazione con le altre tabelle

2.3.2 Redis

Il sistema integra un server redis per la coordinazione tra i vari processi all'interno del backend, per permettere una comunicazione immediata tra i vari processi, gestione della cache delle stats, e la gestione delle websocket Socket.Io integrate per gli aggiornamenti realtime sui client

2.3.3 Gestione funzionalità backend

Il backend di ExploitFarm è composto da 3 parti principali

- HTTP API routing and serving
- Stats Processor
- Submitter Process
- SocketIo Process (skio process)

2.3.3.1 API HTTP

L'API HTTP Si occupa di rispondere alle richieste dei client che per la necessità (come da requisito) di mantenere le configurazioni aggiornate, hanno un peso non indifferente sulle richieste fatte, pertanto le API sono parallelizzate per mantenere il sistema reattivo.

2.3.3.2 Stats processor

Lo stats processor è nato come necessità dato l'elevato costo computazionale del calcolo dei dati per le statistiche inizialmente affidato alle API HTTP. Lo stato processo in maniera asincrona dal resto del sistema, scarica incrementalmente i dati dal database, e incrementa sempre in maniera scalare un dato strutturato json che contiene i contatori di flag e attacchi sulla base di parametri fortemente eterogenei quali lo stato dell'attacco, lo stato della submission della flag, il team target dell'attacco, il client che ha eseguito l'attacco, il servizio associato e l'exploit associato. La seguente struttura dati ha permesso la generazione dei dati ai fini della creazione dei grafici senza appesantire il sistema ad ogni richiesta dei dati. Le statistiche vengono salvati nel server redis permettendone la condivisione tra i vari processi, L'API HTTP si limita a scaricare ed esporre i dati. Ci sono particolari casi per modifica di configurazioni per cui lo stats processor ricrea da 0 le statistiche, ma prelevando i dati a blocchi per evitare sovraccarichi, e un ricalcolo stesso incrementale.

2.3.3.3 Submitter Process

Il submitter process si occupa della submission delle flag al gameserver dell'infrastruttura di gioco. In particolare rispettando i timeout e le regole configurate nella fase di setup e aggiornando i dati con le ultime modifiche eseguite al submitter anche in fase di esecuzione. Il submitter inoltre verifica il corretto funzionamento dello script scritto dal team e segnala tempestivamente sia in tutti i client sia nella pagina frontend eventuali errori o warning permettendo al team di verificare e riparare tempestivamente lo script di submission, evitando di perdere punti a causa di errori nella fase di scrittura dello script di submission.

2.3.3.4 SocketIo Process

Il processo dedicato a socket.io si occupa di replicare su delle websocket accessibili ed usate dal frontend e da xfarm per permettere la segnalazione immediata di cambiamenti (i client poi si occupano di aggiornare i dati che hanno subito aggiornamenti). Inoltre questo processo per gli stessi fini, periodicamente verifica lo stato degli exploit e scatena un evento di aggiornamento per gli exploit in caso un exploit non risponda da troppo tempo. Lo stesso processo si occupa di gestire l'interazione con i vari client negli attacchi condivisi, coordinadoli tramite le websocket socket.io.

2.3.4 Funzionalità frontend

Il frontend deve essere reattivo e dinamico, pertanto utilizza tecnologie come react per la sua realizzazione. Il frontend deve gestire il login alla piattaforma se configurato, eseguire il setup iniziale, offrire un editor per la gestione del submitter, visualizzazione di grafici, tabella delle ultime

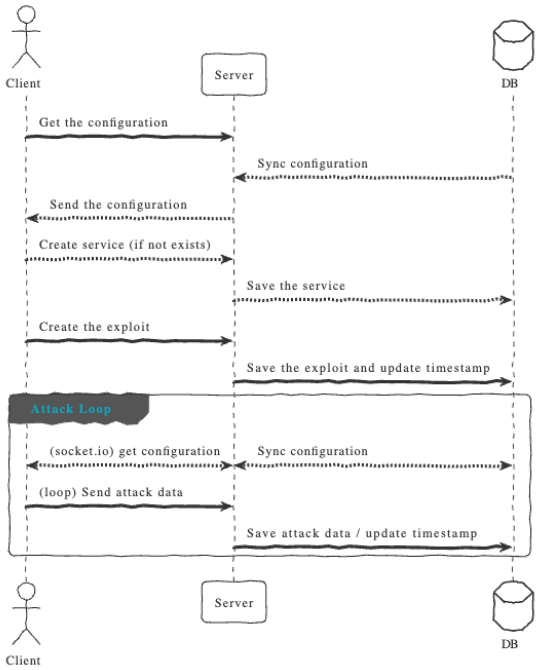
flag attenute, visualizzare lo status dei vari exploit, i log dei vari attacchi. In particolare la schermata principale del frontend deve offrire statistiche tramite grafici su flag, team e attacchi configurabili in base anche a dei filtri nei limiti delle statistiche offerte dal stats processor backend. Tramite le websocket esposte backend il frontend monitora gli eventi e aggiorna i dati che hanno subito modifiche.

2.3.5 Strutturazione del client (xfarm)

Il client xfarm è installato nei sistemi tramite la libreria di exploitfarm disponibile sul package manager di python pip, e consiste in una Terminal UI (TUI) che deve permettere la creazione e avvio degli exploit in maniera semplice. In particolare si deve evitare la memorizzazione e l'utilizzo di flag a riga di comando spesso scomodi da utilizzare. xfarm deve richiedere automaticamente la password di autenticazione se richiesta dal backend stesso, chiedere in input e memorizzare in un file di configurazione a livello di utente di sistema, i dati utili alla connessione al backend di ExploitFarm. Queste informazioni devono essere richieste solo una volta, e il client deve automaticamente rilevare il cambio di server exploitfarm tramite un uuid generato dal server per ogni deploy. Inoltre il client deve assicurarsi che la versione del client stesso coincida con quella del server al fine di evitare incongruenze tra client e server. Il client deve anche creare l'ambiente per l'exploit, dove scrivere l'exploit. Il workspace dell'exploit deve essere adatto alla facile pacchettizzazione e auto-avvio dell'attacco anche su altri client. L'exploit deve poter essere scritto per qualsiasi linguaggio, tuttavia ExploiFarm offrirà funzionalità già scritta nella sua libreria per gli script in python dato che rappresenta il linguaggio maggiormente utilizzato per la scrittura di exploit nelle attack defence.

2.3.6 Autobilanciamento del carico sul client per l'esecuzione degli attacchi

L'esecuzione degli attacchi agli altri team deve avvenire con un timeout per attacco dinamico e calcolato in base alle risorse della macchina su cui il client è in esecuzione disponibili in realtime, quindi deve adattarsi in base al carico che il sistema ha in quel momento. Il flusso di esecuzione di un attacco è rappresentato dal seguente schema:



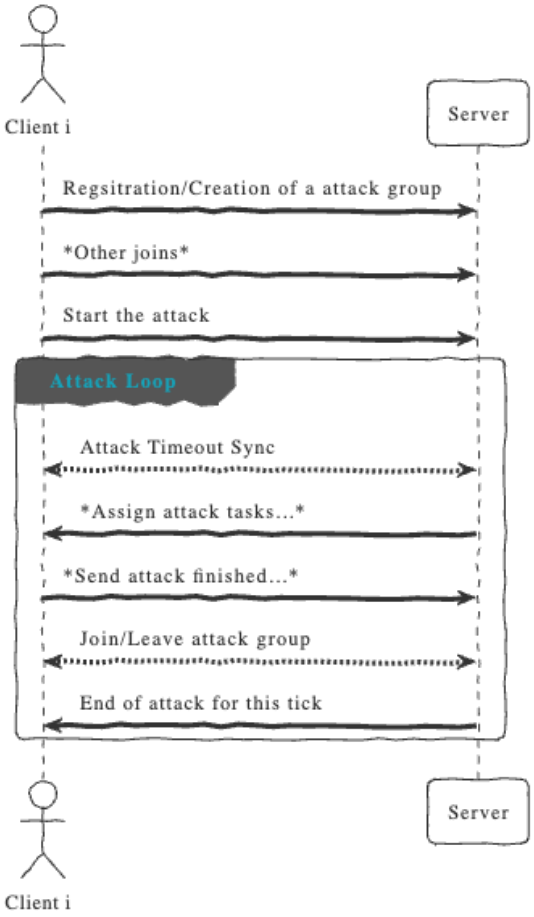
2.3.7 Gestione e versioning degli exploit

Il client deve permettere l'invio il download e l'update degli exploit caricati sulla piattaforma e di quello presente in locale. Questa funzionalità permette una facile condivisione dei sorgenti degli exploit con gli altri componenti del team ed è fondamentale feature per l'implementazione degli attacchi condivisi.

2.3.8 Controllo distribuito degli attacchi condivisi

Il backend tramite anche uno scambio di dati con i client deve bilanciare l'assegnazione dei team ai vari client negli attacchi condivisi senza una configurazione manuale dei pesi sui vari client sulla quantità di team da associare ad ogniuno di questi. Gli attacchi condivisi devono rispettare i cambiamenti di configurazioni e verificare che tutti i client periodicamente, in caso di client non in risposta deve permettere per quanto possibile di portare a termine l'esecuzione di tutti gli attacchi. La comunicazione tra i client e il server riguardante il controllo distribuito è interamente basato su socket.io. Gli attacchi condivisi pertanto procedono secondo il seguente workflow:

1. Creazione di un gruppo di attacco (insieme di client associati ad un exploit)
2. Join di ulteriori client nel gruppo di attacco
3. Start dell'attack group
4. Eventuali nuovi join/left nel gruppo
5. Chiusura del gruppo di attacco



2.3.8.1 Definizione dei parametri disponibili e calcolati dal sistema

Contestualizziamo il problema periodicamente, analizzandone il processo per 1 tick/round, che poi andrà periodizzato per quanto necessario.

- N_C: Numero di client nel gruppo
- N_T: Numero di team totali

  Individueremo in seguito $i = [0, N_C - 1]$ l'indice relativo al client
  Inoltre definiamo $j = [0, N_T - 1]$ l'indice relativo ai team, e quindi all'attacco eseguito nel tick

- Q_i: grandezza della coda di esecuzione per l' i-esimo client. (definita esplicitamente in seguito)
- R_i attacchi in esecuzione sul client (sempre $<= Q_i$)
- T_R Tick/Round time (Tempo definito nel setup relativo alla durata di un round)
- T_K Timeout associato agli attacchi (Kill time)
- $T_{C,j}$ Tempo effettivamente consumato per il j-esimo attacco (per il j-esimo team)
- $T_{T,j}$ Ultimo tempo di timeout calcolato per il j-esimo attacco (per il j-esimo team)

2.3.8.2 Valutazione della potenza computazionale di ogni client

Ai fini di distribuire in base alle risorse disponibili nel gruppo di attacco proporzionalmente a quelle che è la potenza computazionale del singolo client. Al Join/Avvio di un gruppo verrà consigliato un numero di processi in contemporanea da assegnare all'attack group, che sarà personalizzabile dall'utente (questo per valutarlo in base al carico che ci si aspetta di avere per l'attacco). Questo parametro verrà utilizzato come parametro di valutazione sulla potenza del client dato che di default verrà consigliato un valore calcolato sulla base di quanti core sono disponibili sulla macchina.

2.3.8.3 Gestione del tempo di esecuzione per ogni attacco

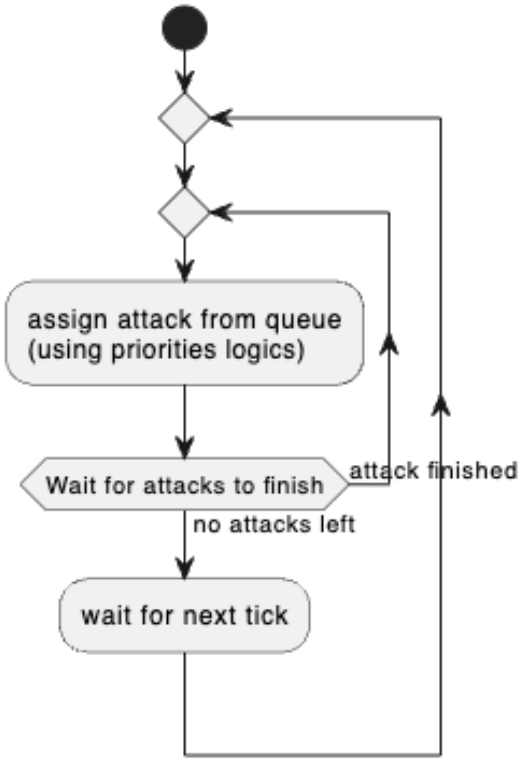
Il sistema ha come obiettivo quello di riuscire ad eseguire tutti gli attacchi entro il tempo di tick: ai fini di perseguire questo obiettivo, si definisce un timeout globale, inizialmente impostato ad un tempo che ne permette l'esecuzione degli attacchi assegnando slot temporali uguali ad ogni esecuzione. Tuttavia durante l'esecuzione degli attacchi, se questi occuperanno meno tempo, dovranno in qualche modo influire sul tempo di timeout incrementandolo, dando quindi la possibilità di terminare l'esecuzione di attacchi su team in cui l'operazione sta richiedendo un tempo superiore a quello medio degli attacchi. Si definisce la tecnica utilizzata per il calcolo di questo tempo dinamico in seguito. Inoltre, i processi seguiranno in ogni caso un timeout globale che ne terminerà l'esecuzione in ogni caso superato il tick time. (Questo timeout è indipendente dal primo timeout dinamico)

2.3.8.4 Algoritmo di distribuzione degli attacchi

Saputi la grandezza delle Q_i code per ogni client, l'assegnazione dei processi avviene tramite una priority queue, in cui ad ogni client viene associato una priorità calcolata, e viene associato un attacco a quel processo che ha priorità maggiore. IL sistema entra in uno stato di attesa se il processo con la priorità più alta ha come valore un valore $<= 0$. La priorità chiamata $P_{CL,i}$ (Processi per client assegnabili) è calcolata tramite la seguente formula:

$P_{CL,i} = \frac{Q_i - R_i}{min\{Q_i\}}$

La seguente formula varia dinamicamente il suo valore per client dipendentemente al carico assegnato, al quantitativo di risorse disponibili e modulato sulla base del client più "debole" nella coda. Nello specifico l'algoritmo prende i client con $P_{CL,i} >= 1$ ed associa ad ogniuno un numero di attacchi pari a $\lfloor P_{CL,i} \rfloor$. Terminata una iterazione sui client, si riesegue la stessa operazione. Se non ci sono client con $P_{CL,i} >= 1$ allora si procede a considerare tutti i client con $P_{CL,i} > 0$ assegnando ciclicamente un processo al client per cui risulta: $P_{CL,i} = max\{P_{CL_i}\}$. Se non ci sono client con $P_{CL} > 0$ allora il sistema rimane in attesa di nuovi eventi. Si specifica che ad ogni assegnazione di un nuovo attacco ad un client R_i viene incrementato: al contrario nel momento in cui un client termina l'esecuzione di un attacco R_i viene decrementato. In caso nuovi client entrano nel gruppo non si modifica lo stato di esecuzione corrente, ma si ricalcolano i coefficienti e si procede come precedentemente descritto. La medesima cosa succede per client che abbandonano il gruppo, che tuttavia in questo caso comporterà un riaccodamento dei processi che erano in esecuzione su quel client. Ciò avviene anche se si verifica la perdita della connessione considerando il client come fuori dal gruppo, in questo caso il client deve terminare la sua esecuzione, poichè il server ri-assegnerà i task che stava eseguendo.



2.3.8.5 Algoritmo di calcolo del timeout

NOTA: Di seguito si descriverà come verrà calcolato il timeout da applicare ad ogni processo dipendentemente dalla possibilità di eseguire tutti gli attacchi entro il tick: Questo timeout quindi definisce il tempo che verrebbe assegnato al primo processo avviato per permetterne l'esecuzione degli altri. Ciò implica che durante i ricalcoli successivi descritti in seguito ci potrebbero essere timeout che eccedono dal tempo del tick. Questo non comporta un problema poichè come descritto precedentemente in contemporanea al vincolo del seguente timeout viene applicato anche un vincolo globale e statico, che mette come soglia massima per l'esecuzione la fine del tick, pertanto non sarà in ogni caso possibile eccedere dal tick per l'esecuzione di un attacco relativo al tick stesso.

Definiamo dei nuovi valori di tempo:

- $T_{J,i}$: Tempo di join dell' i-esimo client: Questo tempo vale 0 se il client ha eseguito il join prima dell'inizio del tick, mentre assume un valore che corrisponde al tempo mancante al termine del tick se riguarda un client che ha eseguito il join durante questo tick.
- $T_{L,i}$: Tempo di left dell' i-esimo client: Questo tempo vale il tempo rimanente lasciato non in esecuzione dalla fine del tick del j-client che esce dal gruppo. Se il j-esimo client non è uscito dal gruppo questo tick questo valore vale 0
- $T_{LJ,i} = T_{J,i} - T_{l,i}$: Tempo di left/join per client (definisce il guadagno totale dato dai left e join di un singolo client).
- $T_{LJ} = \sum_i T_{LJ,j} Q_i$: Tempo totale di left/join (definisce in totale il tempo guadagnato/perso a causa delle operazioni di join e left dei client)
- $T_{G,j} = T_{T,j} - T_{C,j}$: Tempo di guadagno per attacco

- $T_G = \sum_j T_{G,j}$: Tempo di guadagno totale
- $T_{VB} = \sum_i Q_i T_R$: (tempo virtualmente disponibile di base) interpretabile come il tempo totale disponibile di esecuzione su un core singolo per tick, calcolato all'inizio del tick.
- $T_V = T_{VB} + T_G + T_{LJ}$: Tempo virtualmente disponibile

Il tempo virtualmente disponibile tiene conto dei tempi guadagnati grazie alla veloce esecuzione di alcuni attacchi, permettendo di ridistribuire questo tempo tra gli altri processi. Questo tempo può incrementare o diminuire durante l'esecuzione anche in base all' abbandono o al join di nuovi client, pertanto si presta utile al calcolo del timeout flessibilmente a quelle che sono le disponibilità istante per istante.

Tramite questo tempo definiamo infine il timeout $T_K = \frac{T_V}{N_T}$ Anch'esso varia in base al numero di team che potrebbe variare durante l'esecuzione.

3 Project Managment

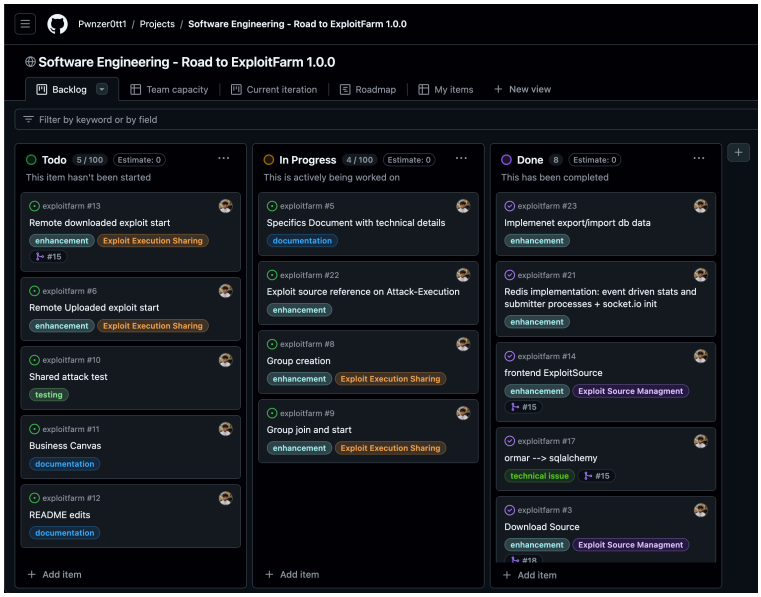
La gestione del progetto è principalmente plan-driven con un approccio però che si avvicina allo "scrum" per progettazione e organizzazione dei task (organizzate in un backlog di user-stories). Il progetto di per se è già in una fase di sviluppo avanzata, pertanto la gestione del progetto riguarderà l'aggiunta della feature per l'avvio di singoli attacchi distribuiti tra più client.

3.1 Gestione generale

L'approccio scelto per lo sviluppo delle nuove funzionalità è basato sullo "scrum" con un approccio tendenzialmente plan-driven dati i tempi brevi assegnati per il progetto stesso. Sono assenti le riunioni giornaliere e tutta l'attività si dividerà in soli 2 sprint per l'integrazione di 2 macro-funzionalità. Sempre a causa di tempi ristretti, i test non sono previsti, ma unicamente micro test durante lo sviluppo stesso per verificarne il funzionamento.

3.2 Kanban (github)

Il progetto è interamente condiviso e gestito su github, grazie alla funzionalità dei progetti di cui si sfrutta il backlog che permette di gestire le user-stories con tag, ordini di priorità e direttamente associabili ai branch in cui si esegue l'attività di sviluppo, per questo molto integrato con lo sviluppo stesso.

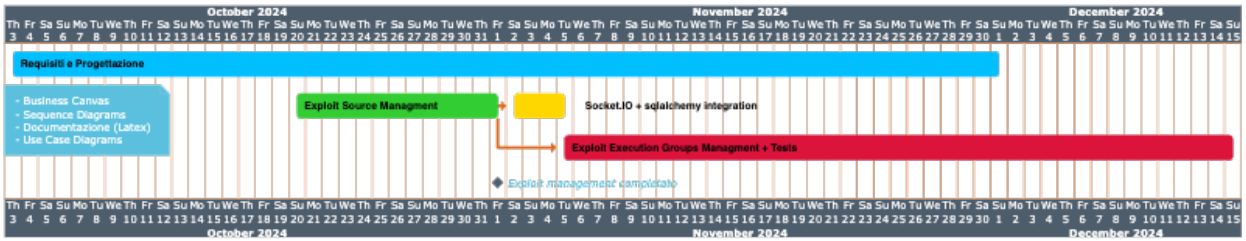


3.3 Scheduling

Sono definite 3 fasi principali nel progetto:

- Brainstorming, definizione dei requisiti tecnici e documentazione
- 1* Milestone/Sprint: gestione degli exploit
- Technical features: From polling to event driven updates + switching to sqlalchemy
- 2* Milestone/Sprint: gestione dei gruppi di client per gli attacchi condivisi

Le fasi sono definite proprio per la necessità di progettare lo sviluppo stesso dei nuovi requisiti, e dalla mancanza della feature di gestione degli exploit necessaria allo sviluppo della seconda milestone. L'inizio del progetto con la prima fase è iniziata il 03-10-2024, la consegna è fissata per il 20-12-2024.



3.4 COCOMO Analysis

Il modello COCOMO utilizzato per la stima è quello del Post-architecture model dato che la fase in sviluppo è una fase di implementazione di feature con un progetto di base già pre-ingegnerizzato (Con size stimata di 2k righe di codice).

$$E = A \times Size^{B} \times \prod_{i=1}^{n} EM_i$$

Con:

$$B = 0.91 + 0.01 \times \sum ScaleFactors$$

- A = 2.94 (tipico per Post-Architecture COCOMO)
- $Size \simeq 2k$ righe di codice

3.4.1 Effort Multipliers

Si utilizzerà un subset dei 17 moltiplicatori previsti per COCOMO post-architecture.

- RELY = 1.26 (Very High)
- DATA = 1.00 (Nominal)
- CPLX = 1.17 (High)
- RUSE = 0.95 (Low)
- DOCU = 1.00 (Nominal)
- TIME = 1.29 (Very High)
- PVOL = 1.00 (Nominal)
- PCAP = 0.88 (High)
- TOOL = 0.90 (High)
- SITE = 1.00 (Nominal)
- SCED = 1.00 (Nominal)

$$\prod_{i=1}^{n} EM_i = 1.431$$

3.4.2 Scale Factors

- PREC = 1.24 (Very High)
- FLEX = 1.01 (Very High)
- RESL = 4.24 (Nominal)
- TEAM = 1.0 (No team, not used)
- PMAT = 1.56 (Very High)

$$B = 0.91 + 0.01 \times \sum ScaleFactors = 0.99$$

3.4.3 Risultato Finale

$$E = A \times Size^B \times \prod_{i=1}^{n} EM_i = 8.36$$

3.4.4 Duration

$$C = 3.67$$

$$D = 0.28 + 0.2 \times (B - 0.91) = 0.296$$

$$T = C \times E^D = 6.8(mesi)$$

Tuttavia assumendo gli obiettivi del progetto, è intuitivo comprendere come la stima eseguita è eccessiva rispetto al reale tempo necessario al raggiungimento dell'obiettivo.

3.5 Risk Managment and Analysis

3.5.1 Stima dell'effort e requisiti

Durante lo sviluppo c'è la possibilità nascano nuove esigenze date da limitazioni tecniche attualmente presenti nel progetto: per attutire i danni possibili generati dalla nascita di nuovi requisiti o anche alla sottostima del tempo necessario, è necessario iniziare al più presto dalla data schedulata l'effettiva data di inizio dello sviluppo di modo da permettere l'allocazione di più slot temporali ad una determinata user-story nello sviluppo prima che possa terminare la deadline di consegna, punto critico nel progetto. Il livello di rischio associato è "Moderato".

3.5.2 Rilascio di versioni non totalmente funzionanti

C'è la possibilità che a causa di modifiche a diverse parti del progetto, si possano presentare bug in parti non direttamente inerenti ma collegate alla parte di software modificata. Per ovviare a questo problema è utile eseguire in maniera atomizzata prima della pubblicazione della release dei test automatici che verificano il funzionamento corretto della piattaforma. Nonostante il rischio si classificato come Medio, date le scadenze strette non è prevista attualmente la scrittura di test automatizzati.

3.5.3 Rischio di rendere la piattaforma complessa gestire

Il progetto ha grandi requisiti e grandi ambizioni a livello di funzionalità che tuttavia dopo essere sviluppate possono risultare complesse nel loro utilizzo anche per gli utenti target che sono classicamente utenti esperti. Andando in questa direzione potremmo perdere uno degli obiettivi del progetto, cioè quello di facile utilizzo, autorizzazione e setup. Il rischio è medio elevato e potrebbe comportare lo sviluppo di parti della piattaforma che necessitano un completo refactoring o riscrittura in seguito. Per evitare questo rischio è necessario un confronto esterno con altri giocatori CTF per avere opinioni esterne, quindi opinioni esenti del bias dello sviluppatore del progetto, che sviluppandone le funzionalità non si rende conto di eventuali funzionalità complesse da utilizzare. Questo permetterebbe una rivisitazione e progettazione migliore del progetto.

3.6 Release Managment

Le release di ExploitFarm avvengono tramite github con una catena di building automatica del container di exploitfarm che contiene frontend compilato e backend, automaticamente rilasciato e pubblicato su gihtub packages che ne rilascia il download pubblico. Inoltre per l'esecuzione di exploitfarm è disponibile uno script che genera dinamicamente un docker compose e lo avvia utilizzando i container pullati dai repository online, e inoltre si assicura che la versione avviata di exploitfarm sia l'ultima disponibile.

3.7 Modello di Business

Il progetto di per se nasce come progetto totalmente opensource, quindi non a scopo di lucro. Tuttavia un possibile approccio di Business adottabile liberamente ispirato a quello di un'altra importante piattaforma opensource per le competizioni CTF Jeopardy (CTFd) è quello di continuare ad offrire una piattaforma completamente opensource e gratuita ma di offrire a pagamento un deploy di ExploitFarm con eventuali client attaccanti hostati (con dimensioni del server adattabili e scalati anche in base al costo e al peso dell'exploit) di modo da rendere immediato l'accesso all'attacker, anche con eventuali supporti rapidi alla connessione a VPN della gara, o al bridging da uno dei PC del team della rete di gara se non sono presenti connessioni VPN.

4 Analisi SWOT

4.1 S: Punti di forza

- Unicità sulle funzionalità
- Interfaccia innovativa
- Setup Semplice
- TUI
- Statistiche

4.2 W: Punti di debolezza

- Elevata complessità
- Centralizzato (SPOF)

4.3 O: Opportunità

- Pubblicizzazione
- Aggiunta di test automatici

4.4 T: Minaccie

- Impossibilità di mantenere il progetto a fronte di una grande complessità da gestire

5 Sviluppi Futuri

- eliminare come single point of failure il server e permettere l'esecuzione di backend multipli e decentralizzati così da evitare in caso di problemi con uno dei server. La gestione di backend decentralizzati è di una complessità altissima e non strettamente necessario per gli obiettivi del progetto pertanto si lascia come sviluppo futuro.
- Sviluppo dei test sulle API backend