Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Sicurezza nei Browser Moderni: Analisi, Minacce e Soluzioni

Relatore:
Prof. Fabio Vitali

Presentata da:
Alessandro Frau

Anno Accademico 2022/2023

Abstract

Questa tesi ha come obbiettivo l'approfondimento dei meccanismi implementati dai browser moderni per mitigare le minacce alla sicurezza web. La trattazione della tesi è strutturata in modo da permettere una visione a tutto tondo dei pericoli affrontati dai browser, introducendo concetti tecnici dietro le vulnerabilità, per poi analizzare attacchi reali, e in conclusione riuscire a mettere in luce come i browser attuino meccanismi di prevenzione e mitigazione delle problematiche. Le principali tematiche trattate nella tesi riguardano il protocollo HTTPS, dove viene esplorata la revoca dei certificati SSL/TLS, il downgrade HTTPS e l'utilizzo di versioni obsolete di SSL/TLS. Riguardo la gestione dei cookie e delle sessioni, vengono analizzati i parametri di sicurezza `SameSite`, `Secure` e `HttpOnly`, per poi passare a i Cookie Prefixes e alle Expiration policy. Viene discussa la Same-Origin Policy e le politiche di Cross Origin Resource Sharing, dove viene posta particolare attenzione sugli attacchi a JSONP, Origin Forgery, l'uso errato della cache HTTP e implementazioni errate delle politiche CORS. Viene anche analizzata l'esecuzione sicura di codice nel browser, e vengono messi in evidenza attacchi comuni come buffer overflow, Use-After-Free e vulnerabilità Javascript. Infine vengono trattati gli elementi di DOM encapsulation, quindi IFRAME e Shadow DOM, con le relative vulnerabilità di Clickjacking, XSS e CSRF. La tesi mostra le tattiche applicate dai browser per affrontare tali minacce, in modo tale da offrire un contributo alla comprensione della sicurezza web.

Indice

Capitolo 1

INTRODUZIONE

Nel panorama moderno, la maggior parte delle attività e interazioni digitali avvengono in rete. Tra i tanti strumenti atti all'utilizzo di internet, lo strumento cardine risulta essere il browser, che quindi si configura come strumento principale per l'accesso a risorse online, ed ha l'enorme responsabilità di essere la prima linea di protezione degli utenti da minacce informatiche in rete. La tesi si propone di esplorare approfonditamente le sfide e le soluzioni che caratterizzano le implementazioni dei browser.

Il presente lavoro si basa su un approccio strutturato, in cui viene fornita una base teorica per i concetti chiave relativi alle vulnerabilità trattate. L'analisi teorica viene successivamente concretizzata attraverso l'esame di vulnerabilità reali, presentate attraverso esempi specifici. L'obbiettivo principale è comprendere come i browser affrontino queste sfide e implementino soluzioni per mitigare o risolvere le minacce emergenti.

Per affrontare al meglio la trattazione estremamente ampia di questa tematica, si è deciso di andare ad analizzare nello specifico solo 4 tra i browser più diffusi alla data corrente, che abbiamo scelto essere Chrome, Safari, Edge e Firefox. Non abbiamo fatto invece distinzione di sistema utilizzato, quindi abbiamo tenuto in considerazione tutti i sistemi operativi per cui questi sono distribuiti. Secondo la piattaforma statcounter, scegliendo questi 4 browser stiamo tenendo in considerazione il 91,62% delle utenze di browser globali [89].

Le vulnerabilità pubblicate per i browser, nonostante il fatto che questi vengano controllati da decenni oramai, vengono comunque scoperte e condivise in maniera piuttosto costante, e fino ad ora non hanno accennato a smettere di essere trovate. Questo può essere influenzato da diversi fattori che riflettono la complessità e l'evoluzione del panorama della sicurezza informatica, tra cui sicuramente il co-

stante aggiornamento delle tecnologie utilizzate e sviluppate. Il seguente grafico, stilato utilizzando le corrispettive liste di fix di vulnerabilità dei browser [75] [22] [7] [64] [63], mette a paragone il numero totale di vulnerabilità trovate per ogni browser, e mostra in maniera chiara come le vulnerabilita che vengono trovate non siano diminuita in maniera sostanziale nel tempo, ma anzi sono rimaste più o meno costanti.

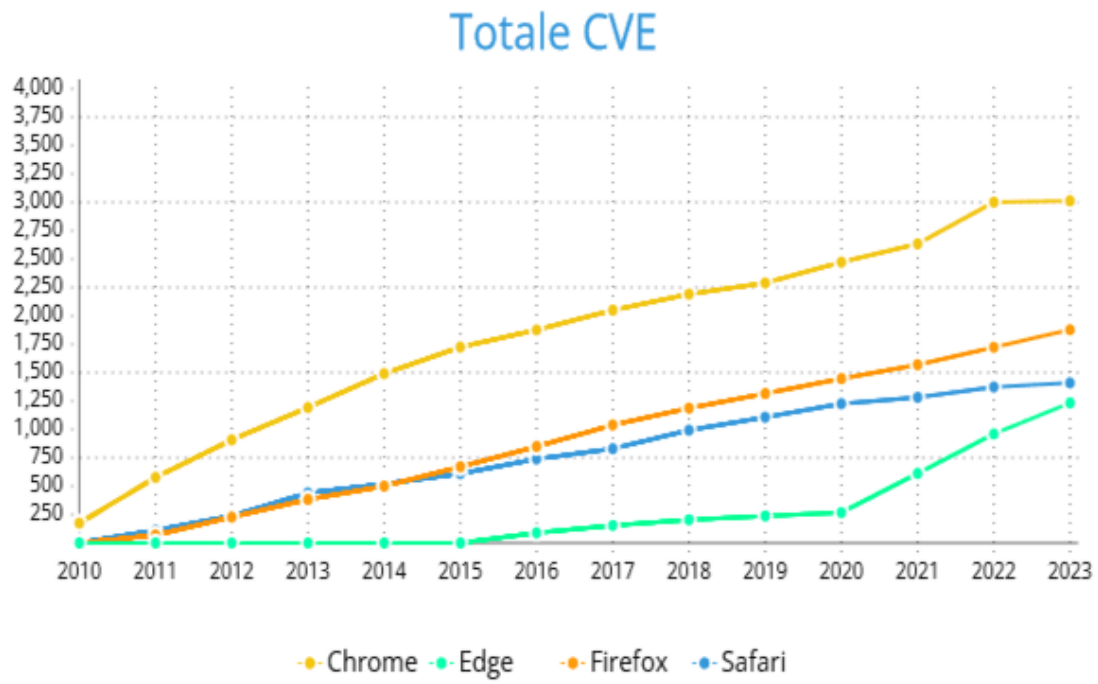

Figura 1.1: Totale CVE per anno

Questa tesi si basa su fonti affidabili e autorevoli, infatti la maggior parte delle descrizioni dei componenti e caratteristiche dei browser sono basate sulle specifiche fornite dal World Wide Web Consortium (W3C) e la Web Hypertext Application Technology Working Group (WHATWG), organizzazioni che sviluppano e mantenengono gli standard per il World Wide Web. Un altra fonte fondamentale di informazioni in questa tesi sono gli RFC (Request For Comments), documenti di specifica tecnica pubblicati dalla Internet Engineering Task Force (IETF). Questi documenti stabiliscono standard, protocolli e linee guida per varie tecnologie internet. Per quanto riguarda le vulnerabilità, durante la tesi viene fatto costante riferimento alle CVE (Common Vulnerabilities and Exposures), un sistema standardizzato per fare riferimento ad una vulnerabilità specifica. Ogni vulnerabilità

è assegnata a un numero univoco (CVE-ID) e catalogata in un database pubblico. Utilizzare le CVE per riferirsi ad una vulnerabilità risulta importante, in modo da evitare fraintendimenti ed essere sicuri di riferirsi in maniera univoca alla stessa vulnerabilità identificandola con un nome. Come piattaforma principale dove andare a visualizzare le informazioni correlate ad un CVE-ID faccio riferimento alla piattaforma del NIST (National Institute of Standards and Technology), un agenzia degli Stati Uniti. Come ultimo, fondamentale, canale di ricerca delle vulnerabilità, sono state utilizzate tutte le piattaforme e i blog ufficiali forniti dai browser, come l'`issue tracker` di chrome [22], i `Security Bulletins` [63] e le `release notes` di Microsoft [64], le `Security Releases` di safari [7] e le `Security Advisories` di Firefox [75], attraverso cui i browser pubblicano tutte le informazioni sugli aggiornamenti di sicurezza effettuati, e dove per i progetti open-source è possibile vedere il processo di scoperta e risoluzione di una vulnerabilità.

Le vulnerabilità possono emergere da diverse fonti, sia retribuite che non, tra cui l'importante gruppo di ricercatori di sicurezza, che tramite le loro competenze tecniche ricercano debolezze all'interno dei software e segnalano le potenziali minacce prima che possano essere sfruttate da malintenzionati. Il loro approccio pro-attivo è essenziale per prevenire attacchi e garantire la robustezza dei sistemi. I programmi di bug bounty stimolano i ricercatori a verificare il livello di sicurezza dei browser, attraverso ricompense finanziarie o altri riconoscimenti. Questa collaborazione aumenta notevolmente la capacità di individuare e risolvere vulnerabilità in modo rapido ed efficace.

Nel capitolo dedicato a HTTPS, l'attenzione è focalizzata sulla comprensione approfondita dell'architettura del protocollo HTTPS e delle tecnologie di sicurezza SSL/TLS. Si esamina il ruolo di HTTPS nella protezione delle comunicazioni in rete attraverso l'implementazione di crittografia robusta. Viene approfondita l'architettura di HTTPS, basata su SSL/TLS, e vengono delineate le differenze tra SSL (Secure Sockets Layer) e TLS (Transport Layer Security), evidenziando il passaggio dall'uso di SSL a TLS a causa delle vulnerabilità rilevate in SSL. Un altro elemento analizzato è la revoca dei certificati SSL/TLS. Si chiarisce il concetto che, nonostante la crittografia forte fornita da HTTPS, un certificato compromesso o non più affidabile deve essere revocato per evitare accessi non autorizzati, e si spiegano le tecnologie CRL e OCSP utilizzate dai browser per mitigare questi attacchi. Il capitolo esamina anche il rischio associato all'uso di versioni obsolete di SSL/TLS, infatti vengono descritti degli attacchi informatici avanzati come POODLE, BEAST, FREAK e Logjam, che hanno sfruttato questo tipo di vulnerabilità. Questa panoramica illustra la necessità di utilizzare le versioni più recenti del protocollo per garantire la sicurezza delle comunicazioni. Infine viene esaminato il rischio di downgrade HTTPS. Questo scenario si verifica quando un attaccante forza una con-

nessione sicura a ridursi a un protocollo meno sicuro, consentendo l'intercettazione delle informazioni trasmesse. Vengono discusse le tecnologie HSTS e le HSTS preload list, utilizzate dai browser per costringere i client a connettersi ai server tramite HTTPS, e prevenire quindi questi attacchi.

Nel capitolo dedicato ai cookie e alla gestione delle sessioni, l'attenzione viene rivolta alla comprensione dell'architettura dei cookie e agli attributi di sicurezza essenziali per preservare la privacy e mitigare i rischi associati. Si esamina l'architettura dei cookie, componenti fondamentali della gestione delle sessioni web. I cookie fungono da meccanismo di archiviazione delle informazioni sul lato client, consentendo alle applicazioni web di mantenere lo stato delle sessioni utente. Successivamente, vengono approfonditi gli attributi di sicurezza applicati ai cookie. Si inizia con l'attributo `SameSite`, progettato per mitigare i rischi di attacchi CSRF (Cross-Site Request Forgery) e migliorare la sicurezza dei cookie. Vengono anche evidenziate delle problematiche legate ai valori di default inconsistenti tra browser e ad implementazioni errate da parte dei browser, sottolineando come a volte non si debba dare per scontata l'efficacia degli strumenti messi a disposizione per la sicurezza. Altro attributo analizzato è l'attributo `Secure`, che specifica che un cookie deve essere trasmesso solo su connessioni sicure, aumentando la sicurezza complessiva delle comunicazioni. L'attributo `HttpOnly` è introdotto come misura di sicurezza per proteggere i cookie da attacchi XSS (Cross-Site Scripting), impedendo l'accesso da parte di script lato client. Questo attributo limita il rischio di furto di cookie e informazioni di sessione. Il capitolo prosegue esaminando l'utilizzo dei cookie name prefixes, un approccio pratico per mitigare la weak integrity dei cookie, e fare in modo che i cookie non possano essere sovrascritti da altri domini. Infine, si sottolinea l'importanza delle expiration policies, che gestiscono la durata di validità dei cookie. Una corretta configurazione di queste politiche è essenziale per migliorare la sicurezza e la privacy, evitando la persistenza eccessiva di informazioni sensibili nei cookie.

Nel capitolo dedicato alla Same-Origin Policy (SOP) e al Cross-Origin Resource Sharing (CORS), l'analisi inizia con un approfondimento dell'architettura di entrambe le politiche, fondamentali per la gestione sicura delle risorse tra origini diverse. Si esamina attentamente la Same-Origin Policy (SOP), mostrando come questa delinea un confine di sicurezza tra le origini, garantendo che il codice eseguito da una pagina web non possa accedere alle risorse di un'altra origine senza autorizzazione. Successivamente, viene approfondito il Cross-Origin Resource Sharing (CORS), che offre un meccanismo per rilassare le restrizioni SOP quando è necessario condividere risorse tra origini diverse. Viene analizzato il funzionamento di CORS attraverso l'introduzione di specifici header HTTP che indicano al browser se una risorsa può essere condivisa con una richiesta da un'origine diversa. In conclusione vengono esplorate alcune vulnerabilità legate a queste politiche. Si esplora come JSONP

possa essere utilizzato come alternativa a CORS per ottenere dati da origini diverse, ma con il rischio di aprire la porta a attacchi XSS. Viene esaminato il rischio di falsificare l'header `Origin` nelle richieste, in maniera tale da eludere le politiche CORS. Si evidenziano le sfide correlate alla gestione della cache HTTP in presenza di richieste CORS, con potenziali problemi di sicurezza, e come queste vengano mitigate dall'header `Vary`. Infine si sottolinea come implementazioni errate di CORS da parte dei browser possano compromettere la sicurezza del sistema, consentendo potenziali attacchi.

Nel capitolo dedicato all'esecuzione sicura di codice nei browser, l'analisi inizia con una comprensione approfondita dell'architettura dei browser moderni, fondamentale per valutare la sicurezza delle esecuzioni di codice all'interno di questo contesto. Si esamina attentamente l'architettura dei browser, parlando anche del modello di processo sandbox che isola le componenti dell'architettura. Questa struttura è progettata per mitigare i rischi associati all'esecuzione di codice potenzialmente dannoso e proteggere l'utente. Successivamente, il capitolo si concentra sugli attacchi alla sandbox del browser, tra cui attacchi di buffer overflow, che sfruttano vulnerabilità nel codice per sovrascrivere la memoria oltre il limite che dovrebbe essere consentito, rendendo possibile l'esecuzione di codice dannoso. Poi viene analizzato il concetto di Use-After-Free, in cui un programma continua ad utilizzare una porzione di memoria dopo che è stata liberata, e di come anche questo concetto venga sfruttato per eseguire codice fuori dalla sandbox del browser. Infine vengono esplorate anche delle metodologie di attacco legate alla sola esecuzione di Javascript nelle pagine web.

Nel capitolo dedicato alla DOM encapsulation, si parte con un'analisi dettagliata delle tecnologie chiave come Shadow DOM e IFRAME, per poi esplorare le vulnerabilità associate a queste implementazioni. Viene spiegato il concetto di Shadow DOM come meccanismo per l'incapsulamento di componenti web, consentendo la creazione di elementi con stili e comportamenti isolati dal resto della pagina. Tuttavia, viene chiarito che lo Shadow DOM non rappresenta una barriera di sicurezza completa e che possono emergere vulnerabilità. Successivamente, il capitolo affronta il ruolo degli IFRAME nella gestione di contenuti isolati all'interno di una pagina. Si esplorano le vulnerabilità legate a Clickjacking, tra cui Clickjacking normale, Cursorjacking e Cookiejacking. Viene sottolineato come gli header `X-Frame-Options` e `Content-Security-Policy` siano utilizzati per mitigare questi rischi limitando l'incorporamento di pagine in IFRAME. Viene esplorato poi il bypass della sandbox degli IFRAME, evidenziando come implementazioni errate da parte dei browser possano compromettere la sicurezza di questo meccanismo di isolamento. Infine si esamina come gli IFRAME possano aprire porte a potenziali attacchi XSS e CSRF.

Come capitolo finale, le conclusioni della tesi raccolgono le sfide affrontate dai bro-

wser moderni nella gestione della sicurezza web e sottolineano le soluzioni implementate per affrontare le vulnerabilità individuate nei capitoli precedenti. La tesi contribuisce alla comprensione della sicurezza web attraverso un'analisi approfondita delle vulnerabilità e delle contromisure implementate nei browser contemporanei.

Capitolo 2

HTTPS

Il protocollo HTTPS *(HyperText Transfer Protocol over Secure Socket Layer)*, anche detto "HTTP Over TLS" è un protocollo definito nell'`RFC 2818`[82], nato per garantire l'Integrità e la riservatezza dei dati comunicati attraverso una rete, grazie all'utilizzo dei protocolli SSL (Secure Sockets Layer) e TLS (Transport Layer Security). HTTPS è utilizzato quindi per rendere la comunicazione di dati sicura nonostante l'utilizzo di un canale non sicuro, proteggendo la comunicazione da attacchi di tipo Man-in-the-Middle[1] (da adesso MITM), eavesdropping[2] e tampering[3]. Tuttavia, nonostante la sua ampia adozione, questo protocollo non è immune da criticità.

Il capitolo vuole esaminare alcune delle vulnerabilità che possono minare l'efficacia del protocollo, mettendo in discussione la cieca fiducia data dall'utente nei meccanismi di comunicazione, e mettendo in evidenza le tattiche applicate dal browser per arginare questi attacchi. Nel contesto costantemente in aggiornamento della sicurezza informatica, è essenziale comprendere potenziali problematiche che si possono celare dietro l'apparente robustezza del protocollo. Attraverso un'analisi approfondita, esploreremo alcune vulnerabilità che potrebbero essere sfruttate dagli attaccanti.

Inizieremo definendo i fondamenti di SSL/TLS, analizzando nello specifico il protocollo di handshake utilizzato. Successivamente, osserveremo alcune vulnerabilità legate a questo metodo di cifratura, esaminando come i browser cercano di arginare

[1] Un attacco Man-in-the-Middle (MITM) è una situazione in cui un attaccante si pone tra due parti comunicanti, intercettando e potenzialmente modificando le informazioni scambiate tra di esse.

[2] L'eavesdropping è l'atto di intercettare conversazioni o comunicazioni private senza consenso.

[3] Il tampering rappresenta l'alterazione o la manipolazione di dati non autorizzata, con l'intento di compromettere l'integrità delle informazioni trasmesse o memorizzate.

le problematiche legate alla revoca dei certificati, come cercano dei compromessi di compatibilità e sicurezza per le versioni obsolete di SSL/TLS, e come gli attaccanti aggirino il browser per effettuare un downgrade del protocollo da HTTPS ad HTTP.

Attraverso l'analisi di studi di caso reali, dimostreremo concretamente come vulnerabilità specifiche abbiano portato a violazioni di sicurezza significative, sottolineando la necessità di una sensibilizzazione alla sicurezza di HTTPS.

2.1 Architettura di HTTPS: SSL/TLS

Per spiegare il protocollo HTTPS in una frase, possiamo riprendere la definizione fornita nell'`RFC 2818`: *"Conceptually, HTTP/TLS is very simple. Simply use HTTP over TLS precisely as you would use HTTP over TCP."* [82].

Il protocollo SSL/TLS è un protocollo che viene posto tra il livello di trasporto e i protocolli applicativi come HTTP e FTP, per rendere la comunicazione sicura. Viene chiamato SSL/TLS perché il protocollo TLS non è altro che una versione più recente del protocollo SSL.

2.1.1 SSL

SSL può essere diviso in due strati: lo strato `SSL Connection`, che lavora a livello di trasporto e si occupa di collegare client e server, e lo strato `SSL Session` che tramite il protocollo di handshaking si occupa di configurare la sessione di connessione. Più nello specifico all'interno del protocollo SSL sono presenti 4 protocolli.

Il protocollo `SSL Record` rende SSL confidenziale e integro, dividendo il messaggio in chunk, comprimendolo (solo dalle versioni di SSL precedenti a SSLv3.0) e cifrandolo con un algoritmo a blocchi o a stream; il protocollo `SSL Change Cipher Spec`, come da nome utilizza un byte per segnalare i cambi di specifiche dello stato di SSL; Il protocollo `SSL Alert`, composto da due byte per segnalare il livello di `alert` e il livello di `severity`, si occupa di segnalare gli errori di comunicazione del protocollo e terminare eventualmente la connessione; Infine abbiamo `SSL Handshake protocol`, che si occupa di gestire la fase di handshake tra client e server, seguendo il seguente schema:

1. Il primo pacchetto viene mandato dal client al server, ed è il pacchetto di `ClientHello`, dove il client invia al server la versione di SSL, un valore randomico, l'id della sessione, il tipo di cifratura utilizzato e la tecnologia di compressione utilizzata;
2. Il server risponde con un `ServerHello`, dove manda le stesse informazioni al client.

3. Il server risponde ancora con il suo certificato X.509 [15], seguito dalla serie di certificati delle certificate authority che hanno generato il certificato del server.

4. Se il server non dispone di un certificato, o ne ha uno utile solo per la firma, manda il messaggio di `ServerKeyExchange`, dove invia tutti i dati necessari allo scambio di chiavi, che variano in base all'algoritmo utilizzato.

5. Un server non anonimo (con certificato), può mandare il messaggio di `CertificateRequest` al client, per richiedere un certificato di autenticazione al client. Nel messaggio viene inviata la lista di tipi di certificato accettati e le certificate authority accettate.

6. Una volta inviati tutti i messaggi precedenti, il server invia un messaggio di `ServerHelloDone`. Una volta ricevuto questo messaggio, il client deve verificare tutte le informazioni ricevute dal server.

7. Il primo messaggio che manda il client al server può essere il `ClientCertificate`, ovvero il certificato del client, che viene mandato solo se il server ha inviato il messaggio `CertificateRequest`.

8. Il client manda il messaggio di `ClientKeyExchange`, dove completa lo scambio di chiavi con il server.

9. Il client manda il messaggio di `CertificateVerify`, dove verifica esplicitamente il proprio certificato.

10. Client e server si scambiano il messaggio `Finished`, per verificare che lo scambio di chiavi e l'autenticazione siano andate a buon fine. Questo è il primo messaggio che viene protetto utilizzando gli algoritmi precedentemente negoziati; client e server possono inviare dati cifrati subito dopo questo messaggio.

2.1.2 TLS

Il protocollo TLS essendo un aggiornamento di SSL funziona in maniera analoga, le modifiche principali sono legate agli aloritmi disponibili nella fase di generazione del MAC, nel protocollo di `SSL Record`, gli algoritmi disponibili per lo scambio delle chiavi, sono stati aggiunti messaggi di alert nel protocollo di `SSL Alert`, e la fase di handshake è stata leggermente ottimizzata.

Alla data corrente gli unici protocolli considerati non deprecati sono TLSv1.2 e TLSv1.3, seguendo la seguente tabella:

Versione	Data di uscita	RFC pubblicazione	Stato
SSLv1.0	Privato	Privato	Privato
SSLv2.0	1995	Draft [78]	Deprecato nel 2011 [80]
SSLv3.0	1996	RFC 6101 [45]	Deprecato nel 2015 [10]
TLSv1.0	1999	RFC 2246 [3]	Deprecato nel 2021 [68]
TLSv1.1	2006	RFC 4346 [41]	Deprecato nel 2021 [68]
TLSv1.2	2008	RFC 5246 [84]	In uso
TLSv1.3	2018	RFC 8446 [83]	In uso

Tabella 2.1: Tabella versioni SSL/TLS

2.2 Vulnerabilità

Nell'affrontare le vulnerabilità di HTTPS, entriamo in una sezione che richiede un'analisi approfondita per comprendere le sfide che possono emergere all'interno di questo protocollo. Mentre HTTPS fornisce crittografia e autenticazione per proteggere le informazioni sensibili, è importante riconoscere che nessuna soluzione è immune da potenziali minacce.

Questa sezione si propone di esaminare le minacce di HTTPS dividendole in tre categorie distinte: la revoca dei certificati SSL/TLS, le debolezze nelle versioni di protocollo e le minacce basate sul downgrade ad HTTP. Attraverso l'esame di casi studio significativi, cercheremo di illustrare con chiarezza le implicazioni pratiche di queste vulnerabilità, come possano impattare la sicurezza delle comunicazioni, e le tattiche applicate dai browser per arginare questo tipo di criticità.

2.2.1 Revoca dei certificati SSL/TLS

Gli attacchi ai certificati SSL/TLS fanno parte di una categoria di minacce che sfruttano falle nelle infrastrutture di certificazione digitale. Può capitare che un attaccante entri in possesso della chiave privata di un certificato, e la utilizzi in maniera impropria, come nel caso `DigiNotar` [44], dove sono stati rilasciati più di 500 certificati falsi dalla CA DigiNotar, o come nel caso di `HeartBleed` nel 2014 [27], dove sono state rubate migliaia di chiavi private dei certificati. Per questo motivo diventa imperativo per le CA avere la possibilità di revocare la validità dei certificati rilasciati, e per i browser avere un meccanismo di controllo della validità dei certificati.

Una volta ottenuta la chiave privata di un certificato, l'attaccante può compiere azioni malevole tramite due vettori d'attacco: ridirezionando l'utente ad un sito di suo controllo, malevolo, tramite un attacco MITM, o impersonificando un sito web

autentico. In base a questi vettori di attacco sono stati implementati due approcci per la revoca dei certificati, tramite Certificate Revocation Lists (da adesso CRL) [15] o tramite Online Certificate Status Protocol (da adesso OCSP) [88].

Le CRL sono liste di certificati revocati, che vengono periodicamente aggiornate dal browser, ma che possono avere grandi dimensioni sul disco, e che comunque non garantiscono la validità del certificato (in quanto questo potrebbe essere stato revocato dopo l'ultimo aggiornamento della lista). L'OCSP è un protocollo che verifica la validità del certificato effettuando una richiesta ai server della CA, ma che può essere facilmente bypassato nel caso il browser implementi una politica `fail-soft`[4], come spesso avviene nei browser moderni per motivi di usabilità, dato che nel caso di attacchi MITM l'attaccante può trivialmente bloccare tutte le richieste del protocollo OCSP.

I browser moderni applicano delle politiche simili tra di loro, ma utilizzano protocolli personalizzati. Chrome utilizza CRLSet [50], che si comporta come una CRL centralizzata gestita da Google, che viene inviata al browser come aggiornamento; Firefox utilizza OneCRL [77], che si comporta in maniera analoga a CRLSet, ma viene gestito da Mozilla. E' facile notare come questo sistema non sia privo di falle, infatti alla data corrente è triviale trovare siti con certificati revocati che non vengono segnalati dal browser.

2.2.2 Versioni SSL/TLS obsolete

Il protocollo SSL/TLS implementa la "Version Rollback Policy": nel messaggio di Hello, viene indicata dai due nodi l'ultima versione di protocollo che supportano, e se le versioni differiscono, selezionano l'ultima versione supportata da entrambi; quindi gli attaccanti cercano volutamente di far effettuare ai nodi un rollback a versioni precedenti alla 1.2, che presentano vulnerabilità, in maniera tale da poter decifrare il traffico. Alcune delle vulnerabilità più note che hanno sfruttato la version rollback policy sono l'attacco POODLE nel 2014 [28], BEAST nel 2011[26], FREAK nel 2015[31] e Logjam nel 2015[33].

L'attacco BEAST (Browser Exploit Against SSL/TLS) sfrutta il downgrade alla versione TLSv1.0, e poi sfrutta il fatto che in questo protocollo l'IV utilizzato dal cifrario a blocchi nel messaggio corrente corrisponde all'ultimo blocco cifrato dal messaggio precedente. Esattamente come in POODLE, l'attaccante è in grado di decifrare un byte del messaggio ogni 256 richieste al massimo.

[4]Con "fail-soft, facciamo riferimento al caso in cui il browser faccia passare il certificato come valido nel caso in cui la richiesta del protocollo OCSP non riceva risposta, al contrario con "fail-hard" ci riferiamo ai casi in cui il browser invalidi il certificato in caso di mancata conferma dal protocollo OCSP

L'attacco POODLE (Padding Oracle On Downgraded Legacy Encryption) sfrutta il downgrade alla versione SSLv3.0, e per una falla nelle implementazioni SSL 3.0, consente all'attaccante di decifrare i messaggi inviati tramite un attacco di tipo padding oracle, che ogni 256 richieste al massimo è in grado di decifrare un byte. Questa vulnerabilità ha portato all'abbandono generalizzato di SSL 3.0 in favore di versioni più recenti di TLS.

L'attacco FREAK (Factoring RSA-EXPORT Keys) sfrutta il downgrade ad una cifratura di livello `export-grade`, sviluppata deliberatamente in maniera debole, introdotta prima degli anni 2000 dalle agenzie governative degli Stati Uniti per decifrare le comunicazioni crittografate straniere. La cipher suite abusata nell'attacco FREAK utilizza RSA con lunghezza della chiave di 512 bit, che rende possibile la fattorizzazione in tempi piuttosto brevi, ad esempio è stato realizzato un servizio che permette la fattorizzazione di chiavi RSA di 512 bit in 4 ore [93].

L'attacco Logjam in maniera simile all'attacco FREAK sfrutta il downgrade ad una cifratura di livello `export-grade`, ma a differenza dell'attacco FREAK, che sfrutta lo scambio di chiavi RSA, Logjam attacca lo scambio di chiavi Diffie-Hellman. Come discusso per la vulnerabilità FREAK le chiavi a 512 bit sarebbero facilmente fattorizzabili, ma secondo l'autore dell'articolo [16] che discute questa vulnerabilità, anche le chiavi a 1024 bit sarebbero fattorizzabili da un attaccante di livello nazionale, e quindi la lunghezza minima consentita per la chiave dovrebbe essere di 2048 bit.

I browser moderni implementano dei meccanismi di sicurezza per prevenire le vulnerabilità citate in precedenza, come evidenzia la seguente tabella.

Vulnerabilità	**Data uscita**	**Fix Firefox**	**Fix Chrome**	**Fix Safari**	**Fix Edge**
POODLE	2014-10-14	2014-12-01	2015-01-21	2014-10-16	-
BEAST	2011-09-06	-	-	2014-02-26	-
FREAK	2015-01-08	-	2015-02-27	2015-03-09	-
Logjam	2015-05-20	2015-07-02	2015-09-01	2015-06-30	-

Tabella 2.2: Data uscita e fix vulnerabilità

Vulnerabilità	**Fix Firefox**	**Fix Chrome**	**Fix Safari**	**Fix Edge**
POODLE	Firefox 34 [71] [70]	Chrome 40 [49]	Safari 6 [4]	Non vulnerabile
BEAST	Non vulnerabile	Non vulnerabile	Safari 6 [81]	Non vulnerabile
FREAK	Non vulnerabile	Chrome 41 [66]	Safari 6 [5]	Non vulnerabile
Logjam	Firefox 39 [74]	Chrome 45 [46]	Safari 8 [6]	Non vulnerabile

Tabella 2.3: Versioni fix vulnerabilità

Al momento, oltre a prevenire gli attacchi citati in precedenza, la maggior parte dei browser principali, non supportano più l'utilizzo di versioni SSL/TLS precedenti a TLSv1.2, eccetto safari.

La seguente tabella mostra la data in cui i principali browser utilizzati al momento hanno reso obsolete le versioni SSL/TLS vulnerabili, precedenti a TLSv1.2.

Browser	**Versione fix rollback policy**	**Data di uscita**
Chrome	Chrome 84 [47]	2018-12-18
Firefox	Firefox 78 [72]	2020-06-30
Microsoft Edge	Edge 84.0.522.40[60]	2020-07-16
Safari	Non deprecato	-

Tabella 2.4: Tabella versioni SSL/TLS risolte

2.2.3 Downgrade HTTPS

In un attacco di downgrade HTTPS l'attaccante tenta di forzare una connessione sicura tra un client e un server a passare da HTTPS a HTTP, riducendo la sicurezza della comunicazione da una connessione crittografata e sicura a una non crittografata e non sicura, per poter leggere ed eventualmente modificare i messaggi trasmessi. La tecnica principale utilizzata per effettuare questo tipo di attacchi è chiamata SSL/TLS stripping [91]; da una posizione di MITM, quando l'attaccante nota che da del traffico HTTP avviene un ridirezionamento a traffico HTTPS, lo elimina in modo trasparente. Cosi facendo La vittima si connette all'attaccante tramite HTTP, e l'attaccante si connette al server tramite HTTPS, trasmettendo i messaggi del client, potendo leggere e modificare il contenuto della comunicazione. Per prevenire questo tipo di attacco è stato implementato il protocollo HSTS (HTTP Strict Transport Security) [53]; il protocollo funziona mediante la risposta del server con un'intestazione speciale chiamata `Strict-Transport-Security`, che segnala al client di utilizzare HTTPS ogni volta che si riconnette al sito. Questa risposta contiene un campo `max-age` che definisce per quanto tempo questa regola deve durare da quando è stata vista per l'ultima volta, solitamente impostata a 2 anni. HSTS presenta una limitazione evidente, in quanto richiede una connessione precedente per informare il browser di connettersi sempre in modo sicuro a un sito specifico. Quando un visitatore si collega al sito per la prima volta, non ha ancora ricevuto la regola HSTS che impone l'utilizzo di HTTPS, quindi un attaccante potrebbe effettuare comunque l'attacco, rimuovendo l'header `Strict-Transport-Security`; per questo motivo sono state implementate le HSTS Preload List, che stabiliscono un elenco fisso di siti che devono essere accessibili solo tramite HTTPS, e vengono

aggiornate all'interno del browser in maniera simile al CRL, e sono implementate nei browser principali come Firefox [73], Chrome [48], Edge [61] e Safari [56].

Capitolo 3

Cookies e Gestione delle Sessioni

I cookie [11] svolgono un ruolo fondamentale nella creazione di un'esperienza personalizzata e nel mantenimento dello stato delle sessioni, e per questo vengono largamente utilizzati dalle applicazioni web odierne. Tuttavia, le loro vulnerabilità a potenziali minacce, come attacchi Cross-Site Request Forgery (da adesso CSRF)[9] e Cross-Site Scripting (da adesso XSS) [62] rendono di fondamentale importanza l'implementazione di misure di sicurezza adeguate.

Il capitolo si propone di esplorare le strategie adottate dai browser per mitigare gli attacchi diretti ai cookie, ponendo particolare attenzione sulle alternative fornite agli sviluppatori per equilibrare la sicurezza con l'usabilità, poiché sono spesso chiamati a trovare un compromesso tra garantire un elevato livello di sicurezza e mantenere un'esperienza utente fluida.

In questo capitolo tratteremo quindi come la sicurezza dei cookie venga implementata tramite alcuni attributi come `SameSite`, che limita l'invio dei cookie solo alle richieste provenienti dalla stessa origine, la flag `Secure` che garantisce la trasmissione dei cookie solo attraverso connessioni sicure (tipicamente HTTPS), e l'attributo `HttpOnly` che impedisce l'accesso dei cookie tramite Javascript. Tratteremo inoltre la pratica del `cookie prefixing`, che contribuisce a minimizzare i rischi associati alla weak integrity, proteggendo ulteriormente da potenziali rischi di lettura non autorizzata, e come la definizione di politiche di scadenza adeguate per i cookie, insieme a timeout di sessione appropriati, limiti la persistenza di sessioni non necessarie, mitigando il rischio di accessi non autorizzati.

3.1 Architettura dei Cookie

I cookie sono piccoli blocchi di dati che svolgono un ruolo cruciale nella gestione dello stato e delle interazioni degli utenti con i siti web. La loro struttura di base consiste in una coppia chiave valore che include metadati significativi come data di scadenza, percorso di validità, dominio associato e vari indicatori di sicurezza.

Il cookie viene solitamente creato dal server, e viene inviato al browser attraverso l'intestazione `Set-Cookie` in una risposta HTTP. Il browser, al ricevimento, memorizza il cookie localmente. Nelle richieste successive verso lo stesso dominio, il browser invia automaticamente il cookie attraverso l'intestazione `Cookie` nella richiesta.

I cookie vengono impiegati in vari scenari, tra cui la gestione delle sessioni utente, dove contengono dati di autenticazione, il tracciamento delle attività di navigazione per analisi comportamentali e pubblicità mirate, nonché la memorizzazione di preferenze utente come lingua e temi.

L'aspetto della sicurezza è fondamentale. I moderni standard prevedono la crittografia tramite il protocollo HTTPS e l'attributo `Secure`, garantendo la trasmissione crittografata di cookie sensibili, poi la specifica `HttpOnly` protegge da attacchi XSS, impedendo l'accesso ai cookie da parte di script lato client, e l'attributo `SameSite` che previene l'invio dei cookie a domini esterni.

Va notato che, nonostante la loro utilità, i cookie sollevano questioni di privacy, poiché possono essere utilizzati per tracciare l'utente. I browser offrono strumenti avanzati di gestione dei cookie, inclusi controlli granulari per la loro gestione. Inoltre, standard come `SameSite` cercano di mitigare potenziali vulnerabilità associate all'uso di cookie di terze parti.

3.2 Attributo SameSite

L'attributo `SameSite` [96] si configura come una caratteristica cruciale nella prevenzione degli attacchi CSRF e nel miglioramento della sicurezza dei cookie. La sua implementazione mira a regolare il comportamento di trasmissione dei cookie, determinando se essi possono essere inviati nelle richieste provenienti da domini esterni. L'attributo può assumere tre valori principali: `Strict`, `Lax` e `None`.

Nel caso di valore `Strict`, il cookie viene trasmesso solo se la richiesta proviene direttamente dalla stessa origine, nel caso di valore `Lax`, invece, permette l'invio dei cookie in determinate situazioni, come nel caso di clic su link esterni, e nel caso di valore `None`, abilita la trasmissione del cookie anche in contesti cross-site, ma richiede l'impostazione del flag `Secure` indicando che il cookie sarà inviato solo su connessioni HTTPS sicure.

3.2.1 Inconsistenza dei valori di default tra browser

Un aspetto rilevante nell'implementazione dell'attributo `SameSite` è rappresentato dalla varietà di valori di default adottati dai diversi browser. La mancanza di uniformità di valore predefinito può generare incoerenze nella gestione dei cookie, causando potenziali vulnerabilità e sfide per gli sviluppatori. Attualmente, Chrome, Firefox e Edge adottano il valore di default `Lax` [21] [69] [65] consentendo la trasmissione dei cookie in situazioni comuni come i clic su link esterni, mentre Safari ha come valore di default `None`. Il metodo migliore per prevenire attacchi dati da queste inconsistenze è quello di configurare in maniera esplicita il valore lato server, infatti anche nei browser dove il valore di default viene impostato a lax, possono esserci delle falle di implementazione, come accaduto nel caso di Google Chrome 98.0.4758.102 (build per ubuntu), dove è stata sfruttata la politica `LAX+POST`[1] per effettuare un attacco di tipo CSRF [17].

3.2.2 Implementazioni errate

L'implementazione dell'attributo `SameSite`, come definito nella specifica `RFC6265bis` [14], genera delle sfide per la conformità dei browser a queste specifiche, introducendo un grado di eterogeneità che può compromettere la coerenza nella gestione dei cookie tra diverse piattaforme. Secondo il paper pubblicato da Soheil Khodayari e Giancarlo Pellegrino a maggio 2022, tra i 14 browser principali che hanno testato, nessuno ha rispettato completamente le specifiche dell'RFC 6265bis[55].

3.3 Attributo Secure

I secure cookies costituiscono una categoria di cookie HTTP dotati dell'attributo `Secure` [11], il quale restringe la portata del cookie esclusivamente a canali considerati "sicuri" (dove la sicurezza è definita dal browser). Quando un cookie è contrassegnato con l'attributo `Secure`, il browser lo include solo in richieste HTTP trasmesse attraverso canali sicuri, quindi tipicamente attraverso HTTPS. Nonostante la sua utilità nel proteggere i cookie da attacchi di rete, l'attributo `Secure` preserva unicamente la confidenzialità del cookie. Un attaccante di rete attivo potrebbe sovrascrivere i cookie sicuri provenienti da un canale non sicuro, compromettendone l'integrità. Tuttavia, alcuni browser, come Chrome a partire dalla versione 52 e Firefox a partire dalla versione 52, decidono di ignorare questa specifica per garantire

[1] La politica LAX+POST prevede che per i primi due minuti in cui un cookie viene configurato senza l'attributo SameSite, viene applicata la politica None, e successivamente viene applicata la politica Lax

una maggiore sicurezza, vietando ai siti non sicuri (HTTP) di impostare cookie con la direttiva `Secure` [76].

3.4 Attributo HttpOnly

L'attributo `HttpOnly` [11] indica al browser di omettere il cookie dalla lista dei cookies, quando questa viene fornita dal browser tramite API diverse da HTTP (o HTTPS). L'obbiettivo principale di `HttpOnly` è mitigare l'efficacia di attacchi XSS, nei quali un aggressore inietta script dannosi in pagine web visitate da altri utenti, spesso con lo scopo di recuperare informazioni sensibili contenute nei cookie (come ID di sessione). `HttpOnly` mitiga gli attacchi XSS in quanto impedisce l'accesso al cookie tramite Javascript. Questa misura protegge da potenziali vulnerabilità lato client, contribuendo a preservare l'integrità delle sessioni utente.

L'attacco più comune ad `HttpOnly` è chiamato Cross-Site Tracing (da adesso XST) [52], in quanto permette ad uno script malevolo di leggere il valore del cookie in risposta ad una richiesta di tipo TRACE [2]. In risposta a questo attacco, i browser hanno iniziato ad impedire agli script di eseguire richieste di tipo TRACE.

3.5 Cookie Name Prefixes

I cookie name prefixes [14] vanno a mitigare la "weak integrity" dei cookie [14], ovvero la possibilità da parte di sottodomini e domini "fratelli" di sovrascriversi i cookie a vicenda. Per mitigare queste problematiche sono stati definiti due prefissi del nome del cookie, infatti se il nome di un cookie ha come prefisso la stringa `__Secure-`, allora il cookie deve essere impostato con l'attributo `secure`, se invece il nome di un cookie inizia con la stringa `__Host-`, allora il cookie sarà stato impostato con l'attributo `Secure`, l'attributo `Path` con valore "/", e nessun attributo `Domain`. Questa combinazione produce un cookie che, per quanto possibile, tratta l'origine come confine di sicurezza, dato che l'assenza di un attributo `Domain` vincola il cookie a un host specifico anziché consentirgli di estendersi a sottodomini. Impostare il `Path` su "/" significa che il cookie è efficace per l'intero host e non verrà sovrascritto per percorsi specifici. L'attributo `Secure` assicura che il cookie non venga alterato da origini non sicure e non si estenda a protocolli diversi. La maggior parte dei browser principalmente utilizzati supporta l'utilizzo dei prefissi nei cookie, in particolare Chrome dalla versione 49, Firefox dalla versione 50, safari dalla versione 17.1 e Edge dalla versione 79 [18].

[2] In una richiesta di tipo TRACE il server risponde riflettendo l'intero contenuto della richiesta nel body (cookies inclusi), che poi può essere letto dallo script malevolo.

3.6 Cookie Expiration Policies

Le politiche di scadenza dei cookie sono uno strumento importante per regolare il periodo di utilizzo di un cookie e sono motivate da considerazioni di sicurezza, privacy e gestione delle risorse. Limitare la durata di un cookie contribuisce a mitigare i rischi di possibili attacchi, poiché un periodo di validità limitato riduce la finestra temporale in cui un potenziale attaccante potrebbe sfruttare il cookie. Dal punto di vista della privacy, limitare la persistenza del cookie aiuta a tutelare le informazioni raccolte dall'utente, evitando associazioni prolungate nel tempo. Inoltre, gestire la durata dei cookie è cruciale per ottimizzare l'uso delle risorse del dispositivo, poiché i cookie persistenti a lungo termine possono occupare spazio di archiviazione. Il cookie può essere impostato con l'attributo `expires` [14], che rappresenta la data in cui il cookie scade, o con l'attributo `max-age` [14], che rappresenta il numero di secondi che devono passare prima che il cookie scada. Nel caso entrambi gli attributi siano definiti, le specifiche impongono che il browser dia priorità all'attributo `max-age`. E' anche specificato che la durata massima di un cookie dovrebbe essere di 400 giorni; questo vincolo al momento viene rispettato solo da chrome tra i browser principali, ma è stato preso in considerazione anche da Firefox e Safari [20].

Capitolo 4

Same-Origin Policy (SOP) e Cross Origin Resource Sharing (CORS)

La Same-Origin Policy (da adesso SOP) [2] [87] [12] e il Cross-Origin Resource Sharing (da adesso CORS) [42] sono due aspetti che definiscono il perimetro di sicurezza nei browser web moderni. La Same Origin Policy è un principio che delinea le regole per le interazioni tra pagine web provenienti da origini differenti. Il suo obbiettivo principale è prevenire attacchi di tipo cross-site, una categoria che comprende minacce come l'iniezione di script malevoli e il furto di informazioni riservate. L'implementazione di SOP consiste quindi nel permettere ad una pagina web di accedere solo alle risorse di un'origine identica a quella della pagina stessa, riducendo drasticamente la possibilità di exploit dannosi. Parallelamente, il Cross-Origin Resource Sharing (CORS) entra in scena per offrire una flessibilità controllata a questa politica. Introducendo header HTTP specifici, CORS consente ai server di dichiarare quali origini sono autorizzate a accedere alle loro risorse. Questo è particolarmente utile con applicazioni web complesse che richiedono il recupero di risorse da origini diverse. Tuttavia, la configurazione accurata di CORS è imperativa per evitare l'abuso di questa flessibilità e garantire che l'accesso alle risorse cross-origin avvenga in modo sicuro e controllato.

In questo capitolo esamineremo prima l'aspetto tecnico delle politiche SOP e CORS, mostrando come queste proteggono gli utenti da attacchi cross-origin ma permettano comunque agli sviluppatori di avere flessibilità. In seguito andremo ad analizzare gli attacchi a questi protocolli, come l'utilizzo di JSONP, che risulta strutturalmente vulnerabile, per poi vedere come tramite implementazioni errate e attacchi alla cache nel passato queste politiche siano state aggirate. Analizzeremo anche come la modifica dell'intestazione `Origin` generi dei problemi per gli sviluppatori.

4.1 Architettura di SOP e CORS

4.1.1 Same-Origin Policy

La Same Origin Policy (SOP) è una politica di sicurezza implementata dai browser per limitare le interazioni tra pagine web provenienti da origini diverse. Essenzialmente, questa politica stabilisce che una pagina web può accedere solo alle risorse (come Javascript, immagini, stili CSS, etc.) di un'origine identica a quella della pagina stessa. Questo principio svolge un ruolo fondamentale nel prevenire attacchi di tipo cross-site, infatti senza SOP, se un utente visitasse un sito non sicuro (quindi qualsiasi sito di cui non si ha letto il sorgente prima della visualizzazione), un attaccante potrebbe fargli eseguire azioni malevole su altri siti.

La SOP si basa sul concetto di `origin`, che è composto da tre componenti principali: schema, cioè la parte iniziale di un URL che specifica il protocollo di comunicazione utilizzato per accedere alla risorsa indicata (ad esempio HTTP e HTTPS), dominio, che rappresenta l'indirizzo IP o il nome di dominio del server che ospita la risorsa, e porta, che specifica la porta del server a cui connettersi. Due pagine web appartengono alla stessa `origin` solo se queste tre componenti coincidono. Ad esempio, due pagine con URL diversi, come https://www.test.com e http://www.test.com, sono considerate origini diverse.

Quando una pagina web A tenta di effettuare una richiesta a un'altra pagina B, il browser applica la SOP per determinare se questa richiesta è consentita. La SOP permette alle pagine di effettuare richieste solo a risorse (come script, immagini o API) che appartengono alla stessa `origin` della pagina. Qualsiasi tentativo di effettuare richieste a risorse di un'origine diversa viene bloccato.

La SOP impone quindi delle limitazioni stringenti, come il blocco delle richieste, che viene applicato nel caso di tutte le richieste effettuate tramite `XMLHttpRequest` o `fetch`, l'accesso ai cookie, che impedisce ad un'origine di accedere ai cookie di un'altra origine, e il divieto di lettura dei contenuti di pagine di origini diversi.

4.1.2 Cross-Origin Resource Sharing

Il Cross-Origin Resource Sharing (CORS) è un meccanismo che consente di rilassare la SOP per consentire le richieste di risorse tra origini diverse. La sua implementazione è fondamentale per garantire che le risorse di un dominio possano essere richieste e utilizzate da un altro dominio in modo sicuro e controllato.

Prima di effettuare la richiesta, il browser effettua una `preflight request`, una richiesta opzionale e di tipo OPTIONS, per verificare se il server consente la richiesta

effettiva. In caso affermativo il browser effettua la richiesta con i metodi e i dati specificati in precedenza.

Il cuore del CORS risiede nelle intestazioni HTTP scambiate tra il client e il server durante una richiesta. Quando un client effettua una richiesta cross-origin, il server deve includere alcune intestazioni specifiche nel suo insieme di risposte per indicare se le richieste da origini diverse sono consentite.

Come prima intestazione abbiamo `Origin`, che viene mandata nella richiesta e che specifica l'origine della pagina. In seguito il server deve rispondere con un insieme di intestazioni CORS.

- `Access-Control-Allow-Origin`, che indica quali origini sono autorizzate a accedere alla risorsa. Questa intestazione può essere un singolo dominio o "*", che indica che tutte le origini sono consentite.
- `Access-Control-Allow-Methods` indica i metodi HTTP (come GET, POST, PUT) consentiti per la richiesta cross-origin.
- `Access-Control-Allow-Headers` specifica gli header HTTP aggiuntivi oltre a quelli di default (come `Content-Type`) che possono essere inclusi nella richiesta cross-origin.
- `Access-Control-Allow-Credentials` indica se il browser deve includere o meno le credenziali (come i cookie) nella richiesta.
- `Access-Control-Expose-Headers` specifica quali header possono essere esposti e quindi visualizzati dal client dopo aver effettuato una richiesta cross-origin.
- `Access-Control-Max-Age` indica per quanto tempo, in secondi, le informazioni sulle autorizzazioni devono essere memorizzate in cache.

Il server può rispondere anche solo con un sottogruppo di queste intestazioni.

E' importante anche definire le `simple-request`, che rappresentano una categoria di richieste HTTP che, per la loro natura, sono considerate sicure e possono essere eseguite direttamente senza la necessità di una `preflight request` preliminare. Questo aspetto del protocollo CORS semplifica il processo di interazione tra risorse di origini diverse quando determinati criteri specifici sono rispettati. Per essere classificata come `simple`, una richiesta deve utilizzare uno dei seguenti metodi HTTP: GET, HEAD, o POST. Questi metodi sono considerati sicuri e non implicano operazioni complesse o potenzialmente pericolose. Le `simple requests` possono includere solo un insieme limitato di intestazioni HTTP, note come intestazioni HTTP semplici. Queste includono, `Accept`, `Accept-Language`, `Content-Language`,

`Range` con valori semplici come `bytes=256-`, `Content-Type` che può avere valori `application/x-www-form-urlencoded`, `multipart/form-data` e `text/plain`. Infine le `simple requests` non includono l'opzione di inviare credenziali (come cookie o header di autorizzazione) con l'intestazione `withCredentials` impostata a `true`. Questo contribuisce a garantire la sicurezza del processo.

La semplificazione introdotta dalle `simple requests` facilita l'interazione tra pagine web e risorse di origini diverse quando le richieste soddisfano le condizioni specifiche stabilite dal protocollo CORS. Questo approccio mira a garantire la sicurezza e il controllo nell'accesso alle risorse cross-origin, promuovendo una pratica gestione delle richieste HTTP.

4.2 Vulnerabilità

Le vulnerabilità associate a SOP e CORS sono spesso oggetto di attenzione da parte della comunità di sicurezza. Ad esempio, uno degli scenari di rischio è l'esposizione di risorse sensibili a una richiesta cross-origin non autorizzata. Di seguito andiamo ad analizzare alcune categorie di attacco

4.2.1 JSONP

JSONP (JSON with Padding) [54] è una tecnica nata quando ancora le intestazioni di `Access-Control` non erano in uso, e veniva utilizzata dagli sviluppatori per aggirare le restrizioni imposte dalla Same-Origin Policy (SOP) nei browser web, consentendo alle pagine web di ottenere dati da server esterni. Sebbene risolva il problema delle restrizioni SOP, JSONP introduce rischi di sicurezza significativi, soprattutto in relazione agli attacchi CSRF (Cross-Site Request Forgery).

Il funzionamento di JSONP coinvolge l'inclusione di un tag `<script>` nell'HTML della pagina web, specificando l'URL di un server esterno come sorgente dello script. L'URL include un parametro che indica la funzione di callback da chiamare. Il server esterno elabora la richiesta e restituisce i dati nel formato di una chiamata di funzione Javascript, noto come `padded JSON` o JSONP. Una volta caricato lo script, la funzione di callback viene eseguita, consentendo alla pagina web di gestire i dati ottenuti.

Tuttavia, la natura di JSONP apre la porta a rischi di sicurezza, soprattutto in relazione agli attacchi CSRF, infatti JSONP può essere sfruttato per eseguire richieste non autorizzate verso risorse che richiedono autenticazione, consentendo a un attaccante di costringere un utente autenticato a eseguire operazioni non desiderate. Inoltre a differenza delle richieste tradizionali, JSONP non incorpora meccanismi di protezione CSRF nativi, rendendo difficile l'adozione di contromisure basate su

token CSRF. Quindi mitigare gli attacchi CSRF con JSONP è complesso a causa della mancanza di supporto per i token di autenticazione nelle richieste JSONP.

I browser hanno attuato varie tattiche per limitare o eliminare questo tipo di attacchi. Come primo elemento, hanno sviluppato nuove tecnologie sicure per sostituire JSONP, come le intestazioni di CORS. Inoltre sono state aggiunte politiche più restrittive per l'invio di cookie in richieste JSONP (l'attributo `SameSite` discusso in precedenza). Questo aiuta a prevenire attacchi CSRF che potrebbero sfruttare JSONP per effettuare richieste non autorizzate a risorse che richiedono autenticazione. In ogni caso JSONP resta una tecnologia considerata non sicura per design, e per questa ragione sempre meno server la supportano.

4.2.2 Origin Forgery

Un presupposto di sicurezza per il Cross-Origin Resource Sharing (CORS) è la presunta impossibilità di falsificare il valore dell'intestazione `Origin` in una richiesta cross-origin. Tuttavia, nella realtà, questa supposizione non sempre si verifica.

L'intestazione `Origin` è stata presa in considerazione come difesa contro gli attacchi CSRF, tuttavia secondo l'`RFC 6454` [12], una richiesta proveniente da contesti sensibili alla privacy dovrebbe avere valore nullo nell'intestazione `Origin`. Le specifiche non definiscono però quando un contesto può essere considerato sensibile alla privacy. I browser impostano valore nullo in vari casi, tra cui le richieste provenienti da pagine locali, e dalle richieste provenienti dagli script all'interno di un IFRAME con attributo sandbox.

Anche il CORS utilizza l'intestazione `Origin`, il problema sorge dal momento che nemmeno negli standard CORS viene fornita una definizione chiara del valore nullo. Un utilizzo comune del valore nullo avviene nel caso in cui gli sviluppatori condividono dati con pagine locali di file (come nelle applicazioni ibride), e utilizzano quindi le intestazioni `Access-Control-Allow-Origin: null` e `Access-Control-Allow-Credentials: true`. Tuttavia, un attaccante può falsificare l'`Origin` con valore nullo da qualsiasi sito web utilizzando la funzionalità sandbox di un IFRAME del browser, come descritto in precedenza. Di conseguenza, questi siti possono essere letti da qualsiasi dominio in questo modo. Jianjun Chen et al. [19] è stato in grado di identificare 3,991 domini con questa configurazione errata.

4.2.3 Cache HTTP

Un altro vettore d'attacco è situato nella cache delle richieste HTTP. Un server infatti può essere interpellato da più siti, e rispondere con un intestazione

`Access-Control-Allow-Origin` diversa a seconda dell'origine che riceve, creando cosi dei disallineamenti. Per ipotizzare una casistica reale, se il server `esempio.com` risponde con un intestazione `Access-Control-Allow-Origin` che ha come valore l'origine impostata nella richiesta HTTP, quando viene contattato da `server1.com` salverà la politica CORS che ha come valore l'origine `server1.com`, e nel caso questa risposta venisse salvata in cache, se un altro server, ad esempio `server2.com`, provasse a contattare `esempio.com` verrebbe bloccato in quanto violerebbe la politica CORS impostata nella richiesta precedente.

Per affrontare questa situazione, il protocollo HTTP fornisce l'intestazione `Vary` [43]. L'intestazione `Vary` viene utilizzata nei protocolli HTTP per indicare ai server proxy e ai browser quale criterio dovrebbe essere utilizzato per determinare se una risorsa in cache è valida per essere utilizzata per una determinata richiesta successiva. In altre parole, l'header `Vary` specifica quali condizioni devono corrispondere affinché la risorsa in cache sia considerata valida per una richiesta successiva.

In questo caso quindi, tramite l'intestazione `Vary: Origin`, il server indica che la risorsa in cache è valida solo se la richiesta successiva proviene dallo stesso dominio di origine (o, nel caso di un valore `null`, se proviene da un contesto di navigazione in incognito o privacy-sensibile). Quindi se una risorsa è stata memorizzata in cache con l'intestazione `Vary: Origin` e una politica CORS specifica per un dominio `esempio.com`, la risorsa sarà utilizzata solo quando la richiesta successiva proviene da un contesto associato a quel dominio.

Va notato che molti sviluppatori non sono consapevoli di questa sottigliezza. Questo può portare a situazioni in cui risorse condivise tra domini possono essere erroneamente memorizzate in cache con una politica CORS incompatibile, causando problemi di accesso. Secondo Jianjun Chen et al. [19] 132,987 domini (il 27% di quelli verificati da loro) non configuravano in maniera corretta la gestione della cache.

4.2.4 Implementazioni errate

L'implementazione errata delle politiche CORS nei browser può derivare da diverse cause, generando inconsistenze e potenziali vulnerabilità. Un motivo comune è l'ambiguità nelle specifiche CORS, che hanno fornito motivo di discussione in varie occasioni [19], portando a comportamenti non uniformi tra browser. Inoltre, nuove versioni dei browser introducono modifiche che potrebbero non essere retrocompatibili, causando problemi con le applicazioni web esistenti. Anche le differenze tra i browser possono influire sull'implementazione di CORS, poiché ognuno ha politiche e implementazioni interne uniche. Queste disparità possono causare confusione per gli sviluppatori che cercano di creare applicazioni web compatibili su più piattaforme. Inoltre, i cambiamenti nelle specifiche CORS nel tempo possono causare

problemi se i browser implementano versioni obsolete o interpretano erroneamente le nuove modifiche. Questo può portare a comportamenti non conformi o indesiderati.

Un caso in cui il browser ha implementato in maniera errata le politiche CORS è quello della CVE-2015-0807 [29], che sfrutta l'implementazione errata della funzione `navigator.sendBeacon` su Firefox, infatti era possibile effettuare un attacco di CSRF creando un server che risponde correttamente alla richiesta di preflight e poi effettua un ridirezionamento (`30x`) al server vittima. Questo oltre a permettere la CSRF, viola le specifiche CORS, che proibiscono il ridirezionamento di una richiesta con preflight.

Un altro caso di implementazione errata nel browser è quello della CVE-2015-3658 [32], dove Safari preservava l'header `Origin` anche nei ridirezionamenti cross-origin, permettendo agli attaccanti di effettuare una CSRF.

Capitolo 5

Esecuzione Sicura di Codice nel Browser

L'esecuzione sicura del codice nel browser è importante per garantire un ambiente online affidabile e protetto per gli utenti. Il concetto chiave dietro questa pratica è l'implementazione di una sandbox [1] del browser, ovvero un contenitore virtuale che isoli il codice eseguito all'interno di una pagina web. Tuttavia, nonostante gli sforzi per creare un ambiente sicuro, la stessa sandbox può essere soggetta a vulnerabilità.

La sandbox opera come un meccanismo di isolamento, impedendo al codice malevolo di interagire con il sistema operativo sottostante o con altri processi. La sua presenza è concepita per prevenire attacchi che potrebbero sfruttare le vulnerabilità nei componenti web, proteggendo così la sicurezza globale del sistema. Tuttavia, quando la sandbox stessa presenta falle, si aprono scenari preoccupanti.

Se un attaccante riesce a eludere la sandbox, può ottenere un accesso non autorizzato alle risorse del sistema operativo e eseguire codice dannoso sul dispositivo dell'utente. Questo può portare a una serie di danni, tra cui il furto di informazioni personali, l'installazione di malware, il danneggiamento dei dati o persino il controllo completo del sistema. Inoltre, un attacco di questo genere potrebbe compromettere la privacy dell'utente.

Il capitolo si propone di analizzare le principali metodologie di attacco dalle sandbox, dando inizialmente una definizione di architettura dei browser, ed in seguito trat-

[1] Una "sandbox" è un ambiente isolato e sicuro in cui eseguire applicazioni o processi senza permettere loro di avere accesso indiscriminato al sistema operativo o alle risorse del computer. L'obbiettivo principale di una sandbox è limitare il potenziale danno che un'applicazione dannosa o un codice non attendibile può causare al sistema.

tando buffer overflow, use-after-free e vulnerabilità Javascript, mettendo particolare enfasi su casi reali.

5.1 Architettura del browser

I browser moderni sono progettati con un'architettura modulare e scalabile, in grado di gestire una vasta gamma di funzionalità e di adattarsi alle mutevoli esigenze degli utenti. E' evidente che browser differenti avranno architetture diverse, quindi di seguito baseremo la nostra descrizione su un'architettura di riferimento proposta da A. Grosskurth e M.W. Godfrey [51] e l'articolo di Paul Irish e Tali Garsiel [79]. Per descrivere un browser ad alto livello possiamo dividerlo in varie componenti.

La componente dell'interfaccia utente di un browser è la parte visibile dell'applicazione con la quale gli utenti interagiscono direttamente. Questa componente offre un'interfaccia grafica che consente agli utenti di aprire e interagire con le pagine web, e di interagire con diverse funzionalità del browser. Ogni browser offre elementi differenti nella propria interfaccia, ma ci sono sicuramente alcuni elementi comuni tra cui la barra degli indirizzi, i pulsanti di navigazione, che consentono agli utenti di muoversi tra le pagine visitate, ricaricarle e interromperne il caricamento, la barra dei segnalibri, che consente agli utenti di organizzare e salvare siti web per un accesso rapido in futuro, le schede, che permettono all'utente di aprire più pagine web contemporaneamente, la barra di stato, che fornisce informazioni sulla pagina web come la presenza di HTTPS o l'avanzamento del caricamento della pagina, e la barra dei menu che contiene opzioni di navigazione e configurazione aggiuntive.

Il motore del browser ha la funzione di intermediario tra l'interfaccia utente e il motore di rendering, coordinando le azioni richieste dall'utente e gestendo il flusso di dati tra i vari componenti del browser.

Il motore di rendering, noto anche come motore di layout, è responsabile dell'interpretazione e della rappresentazione visiva del contenuto delle pagine web. Quando un utente richiede una pagina, il motore di rendering elabora il codice HTML, CSS e XML associato a quella pagina [2], trasformandolo in una visualizzazione che viene mostrata sulla schermata del dispositivo dell'utente. Browser diversi utilizzano motori di rendering diversi, ad esempio Firefox utilizza Gecko, Safari utilizza WebKit, Chrome utilizza Blink e Edge inizialmente utilizzava EdgeHTML, ma poi è passato a Blink [92]. Come prima fase il motore di rendering comunica con la componente network per ottenere il contenuto della pagina, in seguito analizza l'HTML che riceve e genera l'albero DOM, poi analizza gli stili CSS e insieme all'albero DOM generato

[2]Il motore di rendering può elaborare e mostrare altri dati, come ad esempio i PDF, tramite estensioni e plug-in

in precedenza genera l'albero di rendering, che nell'ultima fase detta `painting` viene comunicato alla componente di backend quali componenti devono essere visualizzati a schermo.

La componente di rete si occupa di ottenere tutte le risorse delle pagine che l'utente richiede, effettua quindi le chiamate di rete come le richieste HTTP e HTTPS, e restituisce il contenuto al motore di rendering.

La componente di backend dell'interfaccia utente si occupa di disegnare widget di base come caselle combinate e finestre. Questo backend espone delle API generiche, ma utilizza i metodi dell'interfaccia utente del sistema operativo sottostante.

Il motore Javascript è utilizzato per effettuare un analisi ed eseguire il codice Javascript della pagina, o fornito dal browser (ad esempio attraverso le estensioni), che può interagire con il DOM o il CSSOM. In passato venivano utilizzati degli interpreti Javascript, ma al momento i browser preferiscono la compilazione Just-In-Time [3] per migliorare le performance di caricamento della pagina.

Infine la componente di archiviazione dati viene utilizzata dal browser come strato di persistenza in caso ci sia la necessita di salvare dei dati localmente, come nel caso di cookies, cache e localstorage.

Infine, e di questo tratteremo nella sezione seguente, l'architettura dei browser moderni implementa delle sandbox, in maniera tale da poter analizzare ed eseguire tutto il codice presente nelle pagine in sicurezza. Chrome posiziona i componenti ad alto rischio come il DOM, la Javascript Virtual Machine e il parser HTML all'interno del motore di rendering, che viene messo in una sandbox, per evitare che un attaccante in grado di sfruttare una vulnerabilita del motore di rendering, non possa leggere e scrivere file di sistema dell'utente, nonché eseguire codice arbitrario [1].

5.2 Vulnerabilità della sandbox

Le sandbox, concepite per isolare i processi e garantire un ambiente sicuro di esecuzione, sono costantemente messe alla prova da attaccanti sempre più sofisticati che cercano di sfruttare vulnerabilità. E' importante notare che per sfruttare una vulnerabilità di sandbox escaping l'attaccante deve trovare il modo di effettuare

[3] La compilazione Just-In-Time (JIT) di Javascript è un processo dinamico utilizzato nei motori Javascript dei browser. Durante l'esecuzione del programma, il codice Javascript viene interpretato inizialmente. Successivamente, le porzioni di codice frequentemente eseguite vengono identificate, ottimizzate e tradotte in codice nativo per massimizzare le prestazioni. Questo approccio consente una combinazione di flessibilità interpretativa e esecuzione ottimizzata, migliorando l'efficienza complessiva del codice Javascript.

l'exploit, che spesso comporta l'esecuzione arbitraria di codice tramite componenti malevoli installati dall'attaccante nel browser, oppure qualche meccanismo in grado di superare i meccanismi di sicurezza imposti dal motore di rendering; questo passaggio intermedio rende il processo esponenzialmente più complicato in quanto i browser implementano meccanismi di sicurezza all'interno del motore di rendering e dei componenti esterni che vengono installati, ma come vedremo in seguito non lo rendono impossibile. Avviene più spesso di quanto si pensi che questo tipo di attacco venga scoperto, ad esempio nell'ultimo anno solo su Chrome sono state pubblicate 18 CVE che permettono di effettuare un attacco di sandbox escaping [38]. possiamo osservare come i browser abbiano avuto una quantità considerevole di questo tipo di vulnerabilita, ad esempio prendendo alcuni tra i browser più famosi.

Browser	**Numero CVE**
Google Chrome	103 [38]
Firefox	17 [40]
Microsoft Edge	6 [39]

Tabella 5.1: Tabella di CVE di sandbox escaping nei browser

Esistono vari tipi di attacco che portano ad un escape della sandbox, di seguito andiamo ad analizzarli prendendo come esempio degli attacchi noti del passato.

5.2.1 Buffer Overflow

Gli attacchi di tipo buffer overflow consistono nel sovrascrivere delle aree di memoria per ottenere l'esecuzione di codice dannoso. Un esempio di buffer overflow che ha portato ad un attacco di sandbox escape è la CVE-2023-36719 [37], dove è stato scoperto che in una libreria di Windows vulnerabile, interpellabile tramite la `Web Speech API` di tutti i browser Chromium, tramite Javascript, era possibile effettuare un attacco di tipo buffer overflow e quindi eseguire codice arbitrario. Questo è stato possibile in quanto Chromium non ha un motore di TTS proprietario, quindi quando viene utilizzata l'API di Javascript il browser interpella una interfaccia di sistema (che viene eseguita fuori dalla sandbox), chiamata `ISpVoice` nel caso di Windows. Questa interfaccia carica una DLL vulnerabile a buffer overflow, e quindi apre la possibilità di effettuare un attacco di sandbox escape [58]. Questo è anche uno dei pochi casi dove è possibile effettuare una sandbox escape direttamente da Javascript, senza compromettere il motore di rendering o caricare componenti malevoli.

5.2.2 Use-After-Free (UAF)

Gli attacchi di tipo Use-After-Free consistono nell'utilizzare un riferimento ad un oggetto di memoria liberato, consentendo quindi di eseguire codice arbitrario. Un esempio di Use-After-Free che ha portato ad una sandbox escape è quello della CVE-2021-30528 [35], che sfrutta l'auto completamento delle carte di credito su Chrome per android. Quando il browser cerca di ottenere i dettagli della carta di credito per l'auto completamento, viene chiamato un metodo denominato `IsUserVerifyingPlatformAuthenticatorAvailable`, che su android richiama un metodo di Java chiamato `InternalAuthenticator`. Questo metodo salva `mNativeInternalAuthenticatorAndroid` in un callback come funzione lambda di Java. A questo punto `mNativeInternalAuthenticatorAndroid` distrugge `InternalAuthenticator` quando viene distrutto lui stesso, ma la funzione lambda ha mantenuto un riferimento alla zona di memoria di `InternalAuthenticator`, e lo tiene quindi in vita mentre `mNativeInternalAuthenticatorAndroid` punta ad un oggetto liberato che può essere utilizzato. Questa vulnerabilita può essere sfruttata per ottenere un escape della sandbox, ma per essere sfruttata necessita o che l'utente abbia una carta di credito salvata, o che l'attaccante abbia compromesso prima il motore di rendering [67].

5.2.3 Vulnerabilità Javascript

Può capitare che tramite delle vulnerabilità nel motore di Javascript del browser sia possibile effettuare un attacco di sandbox escape, come nel caso della CVE-2022-1529 [36] su Firefox, che sfrutta una prototype pollution nel processo con esecuzione Javascript privilegiata per eseguire codice arbitrario, e che quindi permette di effettuare una sandbox escape, ad esempio disabilitando la sandbox una volta ottenuta l'esecuzione di codice [57]. Per sfruttare questa vulnerabilità l'attaccante deve prima compromettere il motore di rendering.

Capitolo 6

DOM encapsulation

La pratica di incapsulare il Document Object Model (DOM) [97] è diventata una componente comune nella difesa contro una sempre più ampia gamma di minacce. La DOM encapsulation, o incapsulamento del DOM, è una metodologia di sviluppo che si propone di proteggere il DOM di una pagina web, limitando l'accesso e manipolazione indesiderati da parte di script esterni. Questa pratica è implementata attraverso diverse tecnologie, tra cui Shadow DOM [97] e IFRAME[98].

Il capitolo si propone di affrontare le sfide che i browser affrontano relative alla DOM encapsulation, mettendo in evidenza i metodi utilizzati dagli attaccanti per eludere i meccanismi di sicurezza implementati dai browser e ottenere i dati protetti dal DOM, e mettendo anche in evidenza come queste tecnologie portino a tutta una serie di attacchi collaterali come il Clickjacking.

Il capitolo si sviluppa analizzando come prima cosa le tecnologie principali citate in questo capitolo, quindi DOM, IFRAME, e Shadow DOM. In seguito analizza come nel tempo gli attaccanti hanno abusato e bypassato queste tecnologie in vari modi, come nel caso degli IFRAME, che vengono posizionati in maniera fraudolenta e trasparente sopra ad altri elementi negli attacchi di Clickjacking in modo tale da far cliccare un elemento nascosto all'utente, a sua insaputa, spesso portando a azioni non desiderate o il furto di informazioni sensibili; analizzeremo anche lo sfruttamento degli IFRAME per effettuare attacchi XSS e CSRF, e vedremo come gli attaccanti sono riusciti in passato a eluderne la sandbox per accedere ai dati sensibili all'interno. Infine analizzeremo lo Shadow DOM, e come la sua encapsulation possa essere superata per ottenere informazioni sensibili.

6.1 Architettura DOM, Shadow DOM e IFRAME

Il DOM non è solo un concetto astratto; è il tessuto connettivo che abbraccia ogni elemento di una pagina web. La sua funzione primaria è fornire un'interfaccia di programmazione standard che consente agli sviluppatori di manipolare dinamicamente il contenuto, la struttura e lo stile di una pagina utilizzando linguaggi di scripting, principalmente Javascript. Il DOM agisce come un rappresentante oggettivo del documento, traducendo il codice HTML o XML in una struttura facilmente manipolabile.

La struttura del DOM è organizzata come un albero gerarchico, con un nodo radice che rappresenta l'intero documento. Ogni elemento HTML o XML è rappresentato come un nodo nel DOM, e i nodi figlio rappresentano gli elementi nidificati. Questo approccio gerarchico riflette la struttura innata del documento e consente una rappresentazione chiara e organizzata.

Quando un browser carica una pagina, inizia il processo di parsing del documento, costruendo il DOM dal codice HTML presente. Una volta che il DOM è stato creato, gli sviluppatori possono accedere e manipolare i nodi utilizzando Javascript. Metodi come `getElementById`, `getElementsByClassName` e `querySelector` consentono di selezionare specifici elementi nella pagina.

La capacità di modificare dinamicamente il DOM attraverso operazioni di aggiunta, rimozione o modifica di nodi è fondamentale. Questo dinamismo consente agli sviluppatori di rispondere agli eventi utente o di aggiornare il contenuto della pagina in tempo reale.

Il DOM è un ambiente event-driven, dove eventi vengono generati in risposta alle azioni dell'utente o a cambiamenti nello stato del documento. Questo rende possibile la creazione di applicazioni interattive e reattive che rispondono dinamicamente alle interazioni dell'utente.

Un altro principio importante è la separazione tra struttura (HTML) e presentazione (CSS). Questa separazione permette uno sviluppo modulare e la gestione efficiente del codice.

Con il proliferare delle applicazioni web, soprattutto quelle single-page e ricche di interazioni, la complessità del codice Javascript e delle interfacce utente è notevolmente aumentata. Gli sviluppatori si sono trovati a gestire una grande quantità di codice, spesso distribuito in modo globale, con il rischio di interferenze tra diverse parti dell'applicazione. Questa mancanza di isolamento può portare a collisioni di nomi, ambiguità nel comportamento degli script e difficoltà nella manutenzione.

Con l'aumento della richiesta di componenti web riutilizzabili, è diventato fondamentale garantire che ciascun componente mantenga la sua integrità e indipendenza dal resto dell'applicazione. In assenza di una pratica di isolamento, la realizzazione di componenti riutilizzabili comporta il rischio di conflitti e interferenze con lo stile e il comportamento di altri componenti o del DOM globale. L'introduzione delle Web Components [95] come un insieme di tecnologie, tra cui lo Shadow DOM, HTML Templates, Custom Elements e HTML Imports, ha fornito un modo standardizzato per affrontare questa sfida. Le Web Components consentono agli sviluppatori di creare componenti isolati, ciascuno con il proprio DOM incapsulato e stile separato, riducendo così il rischio di collisioni e semplificando la gestione del codice.

Parallelamente alle Web Components, l'elemento IFRAME si è rivelato una tecnologia preziosa per incapsulare contenuti di terze parti. Gli IFRAME consentono di incorporare un documento HTML all'interno di un altro, creando un ambiente separato. Questo approccio è spesso utilizzato per integrare widget esterni o contenuti senza compromettere la sicurezza e l'integrità del DOM principale.

La DOM encapsulation contribuisce quindi a mantenere la separazione tra struttura e presentazione. Ciò significa che le modifiche apportate a uno specifico componente non dovrebbero avere impatti inaspettati su altri componenti o sull'aspetto globale dell'applicazione. Questa modularità e indipendenza sono fondamentali per lo sviluppo e la manutenzione agili delle applicazioni web. Un'altra motivazione importante per l'adozione della DOM encapsulation è la sicurezza delle applicazioni web. La pratica di isolare il DOM contribuisce a mitigare le vulnerabilità come Cross-Site Scripting (XSS), in quanto limita l'accesso a parti critiche del DOM da parte di script esterni non fidati. Questo è particolarmente rilevante quando si incorporano contenuti di terze parti tramite IFRAME o altri meccanismi.

6.1.1 IFRAME

Gli IFRAME forniscono una soluzione elegante per incorporare contenuti da fonti esterne all'interno di una pagina. Per comprendere a pieno il loro ruolo, vogliamo esplorare come gli IFRAME siano impiegati per incapsulare e integrare porzioni specifiche di contenuto, garantendo al contempo sicurezza e modularità.

L'elemento IFRAME consente di incorporare un documento HTML all'interno di un altro. Questa caratteristica è ampiamente utilizzata per integrare contenuti di terze parti all'interno di una pagina web. Ad esempio, potrebbe essere impiegato per un video, un widget di un social media, o anche per creare contenuti dinamici all'interno di una pagina senza dover ricaricare completamente la pagina stessa. Ad esempio, un'applicazione web potrebbe utilizzare IFRAME per visualizzare ante-

prime di documenti, mappe interattive o widget di calendario senza interrompere l'esperienza utente principale.

Un altra caratteristica degli IFRAME è la loro capacità di creare un ambiente isolato. Il contenuto all'interno di un IFRAME è trattato come un documento separato con il proprio contesto di esecuzione. Ciò significa che lo stile e lo script all'interno dell'IFRAME non interferiranno con il DOM principale della pagina, contribuendo a prevenire collisioni e conflitti, e per questo motivo giocano un ruolo critico nella sicurezza delle applicazioni web. Tuttavia, vedremo in seguito che gli IFRAME devono essere utilizzati con attenzione, poiché possono presentare rischi come il Clickjacking o il bypass dell'incapsulamento dell'IFRAME.

6.1.2 Shadow DOM

Lo Shadow DOM si configura come uno strumento utile ad affrontare le sfide legate all'incapsulamento e alla gestione avanzata dei contenuti. Come parte delle Web Components, lo Shadow DOM offre un approccio flessibile e potente per isolare parti del DOM, contribuendo alla creazione di componenti riutilizzabili e altamente modulari.

L'utilizzo dello Shadow DOM inizia attraverso Javascript, grazie al metodo `attachShadow`, che permette di creare un sotto-albero DOM separato all'interno di un elemento HTML. Questo sotto-albero, detto "Shadow Tree", è isolato dal DOM principale dell'applicazione, che viene chiamato "Shadow Host". Ciò significa che gli elementi, gli stili e gli eventi definiti all'interno di uno Shadow DOM non interferiscono con il resto della pagina e viceversa. Lo Shadow DOM può essere "aperto" o "chiuso" in base alle esigenze dell'implementazione. Questo concetto di apertura o chiusura si riferisce all'accessibilità dello Shadow DOM dall'esterno, ovvero dal codice esterno all'elemento host. Nel caso in cui sia chiuso, il suo contenuto e la sua struttura sono inaccessibili dall'esterno. Nessun altro script al di fuori del componente può manipolare direttamente il suo Shadow DOM. Questo garantisce l'incapsulamento. Nel caso sia aperto, è possibile accedere al suo contenuto e modificarlo da script esterni. Tuttavia, l'accesso diretto allo Shadow DOM comporta il rischio di interferenze e compromettere l'incapsulamento del componente.

L'utilizzo dello Shadow DOM conferisce quindi dei benefici considerevoli, come l' incapsulamento, in quanto consente di isolare completamente la struttura di un componente, rimuovendo le interferenze con altri elementi della pagina, la riutilizzabilità, dato che le entità sviluppate risultano essere indipendenti e facilmente integrabili in diverse parti della tua applicazione, senza preoccupazioni di possibili collisioni con altri elementi, e un ottima gestione degli stili, in quanto permette di applicare stili in modo isolato, senza che essi si propaghino incontrollatamente

al di fuori del componente, consentendo una gestione più precisa dell'aspetto visivo, contribuendo a una migliore manutenibilità del codice e prevenendo conflitti indesiderati.

6.2 Vulnerabilità

Gli IFRAME e lo Shadow DOM sono entrambi strumenti potenti ma suscettibili a possibili minacce. Esaminiamo alcune vulnerabilità rilevanti, evidenziate tramite CVE, e le rispettive soluzioni implementate nei browser moderni.

Gli IFRAME hanno mostrato vulnerabilità come il Clickjacking e la possibilità di attacchi XSS. Per mitigare queste minacce, i browser hanno introdotto l'header HTTP X-Frame-Options, consentendo ai siti di definire quali pagine sono autorizzate a incorporare i loro IFRAME. Parallelamente, l'implementazione della "sandbox" all'interno degli IFRAME ha rafforzato la sicurezza, limitando le azioni possibili e riducendo il rischio di attacchi XSS.

Lo Shadow DOM ha anch'esso affrontato sfide come la potenziale escalation dei privilegi e la possibile leakage di informazioni sensibili. I browser hanno risposto a tali problematiche implementando politiche di sicurezza più rigorose, definendo chiaramente le restrizioni sull'accesso e garantendo una migliore protezione contro le interazioni possibili con lo Shadow Host.

6.2.1 Clickjacking

Il Clickjacking, conosciuto anche come UI Redressing [59], sfrutta la trasparenza degli IFRAME per ingannare gli utenti e ottenere il loro consenso involontario a eseguire azioni non desiderate.

In un contesto di Clickjacking, gli attaccanti sovrappongono elementi trasparenti o invisibili sopra contenuti legittimi, inducendo gli utenti a fare clic su qualcosa che non intendono. Questo può portare a una serie di azioni indesiderate, come il consenso a transazioni non autorizzate, la condivisione di informazioni sensibili, o l'attivazione di funzionalità senza consapevolezza dell'utente.

Con il tempo sono stati ideati vari tipi di attacco di Clickjacking, a partire da quello classico, che i browser cercano di prevenire con dei meccanismi di rilevamento automatici, ma a volte falliscono come nel caso della CVE-2019-5861 [34], in cui Chrome non rileva che due elementi sono sovrapposti se la sovrapposizione avviene tramite la trasformazione di CSS `scale`. Un altro tipo di attacco è detto Cursorjacking, e consiste nel modificare l'aspetto del cursore del mouse su una pagina web per ingannare gli utenti e farli cliccare su elementi non desiderati. Questo tipo di manipolazione

visiva può portare a azioni non autorizzate o al consenso involontario dell'utente a operazioni dannose. Un esempio di mancata protezione del browser al Cursorjacking è la CVE-2015-0810 [30], dove in Firefox per MacOSX è possibile rendere il cursore invisibile tramite un oggetto di flash e un div trasparente. Un altra minaccia è il Cookiejacking [94], in cui è possibile ottener un accesso non autorizzato ai cookie del browser dell'utente, facendogli scorrere un oggetto apparentemente innocuo sullo schermo, ma che in realtà seleziona il contenuto del cookie, che viene poi copiato e inviato all'attaccante. Un esempio di Cookiejacking è quello della CVE-2011-2382 [25], dove su internet explorer è possibile inserire il contenuto del file dei cookie del browser all'interno di un IFRAME, che poi viene fatto selezionare all'utente rendendo l'IFRAME invisibile, per poi copiare il testo e rubare i cookie.

Per contrastare queste minacce, i browser moderni hanno adottato misure preventive. L'header `X-Frame-Options` [85] è uno strumento che consente ai siti di definire quali pagine sono autorizzate a incorporare i loro IFRAME. L'opzione `SAMEORIGIN` impedisce l'incorporazione dell'IFRAME da parte di pagine esterne, riducendo notevolmente il rischio di Clickjacking, mentre l'opzione `DENY` blocca l'utilizzo degli IFRAME nel sito. Altre opzioni come `ALLOW-FROM`, offrono ulteriori livelli di controllo sulla politica di incorporazione. Un' altro strumento che viene utilizzato è l'opzione `frame-ancestor` dell'header `Content-Security-Policy` [23], che permette di limitare gli URL utilizzabili dagli IFRAME come risorsa. Con l'aggiunta dell'opzione `frame-ancestor` nell'header `Content-Security-Policy`, l'header `X-Frame-Options` è tecnicamente deprecato [24], ma i browser continuano a supportarlo per retro compatibilità.

Ulteriori contromisure includono l'implementazione di script di difesa lato client, come il `frame-busting`, che rileva e risponde alle tentativi di incorporare una pagina in un IFRAME non autorizzato, ma nel tempo vengono sempre meno utilizzati in quanto si sono dimostrati in larga misura inefficaci contro attaccanti abbastanza determinati [86].

6.2.2 IFRAME sandbox bypass

Il bypass della sandbox degli IFRAME è un tentativo di eludere le restrizioni di sicurezza imposte dall'attributo sandbox in un elemento IFRAME. L'attributo sandbox è utilizzato per limitare le azioni consentite all'interno dell'IFRAME, riducendo il rischio di attacchi come l'esecuzione di script dannosi o la visualizzazione di contenuti non sicuri.

Tuttavia, i tentativi di bypass della sandbox pongono una minaccia all'integrità di queste restrizioni. Una vulnerabilità di questo tipo è stata dimostrata da Aidil Arief [8], che ha scoperto un'implementazione errata della funzio-

nalità `allow-same-origin` nella sandbox degli IFRAME di Firefox. Il valore `allow-same-origin` permette all'IFRAME di non fallire la Same-Origin Policy, e quindi di accedere a tutti i dati della pagina come cookies, localstorage e alcune API di Javascript. Gli IFRAME hanno un attributo detto `srcdoc`, che consente di specificare il contenuto HTML dell'IFRAME direttamente nel tag, fornendo una fonte di dati interna senza dover fare riferimento a un URL esterno. Se all'interno di `srcdoc` inseriamo un XSS con il tag `<a>`, ad esempio:

```
<a href='Javascript:alert(1)'>CLICK</a>
```

Possiamo far avvenire l'XSS se premiamo sul tag con la rotellina del mouse, o se facciamo `CTRL+click`.

6.2.3 XSS e CSRF tramite IFRAME

L'integrazione di IFRAME all'interno delle pagine web, sebbene una pratica comune e versatile, introduce potenziali rischi di sicurezza, con minacce prominenti come Cross-Site Scripting (XSS) e Cross-Site Request Forgery (CSRF) che possono essere sfruttate attraverso questo elemento. L'inclusione di IFRAME apre la porta a rischi di XSS, che possono manifestarsi in diverse forme. Nel contesto di reflected XSS, gli attaccanti possono inserire script dannosi come parte dell'indirizzo o della richiesta, rischiando di riflettersi nell'IFRAME e compromettere la sicurezza della sessione. Allo stesso modo, l'IFRAME può essere strumento di stored XSS, consentendo agli attaccanti di iniettare script malevoli nei dati salvati, potenzialmente portando a una compromissione persistente.

Gli IFRAME possono anche essere sfruttati per condurre attacchi CSRF. Nel caso di login CSRF, un IFRAME malevolo potrebbe tentare di eseguire azioni autenticandosi automaticamente con le credenziali dell'utente, sfruttando la sessione autenticata, oppure, attraverso richieste ad immagini o risorse, gli attaccanti possono utilizzare IFRAME per inviare richieste che implicano azioni non desiderate, approfittando della sessione autenticata dell'utente.

La responsabilità primaria per la prevenzione di queste categorie di attacchi ricade sugli sviluppatori delle applicazioni web, in quanto i browser forniscono misure preventive fondamentali, tramite gli header di sicurezza, ma queste devono essere implementate e configurate correttamente dagli sviluppatori per garantire una protezione efficace.

6.2.4 Sicurezza nello Shadow DOM

Lo Shadow DOM emerge come una tecnologia potente per l'incapsulamento di componenti e stili, contribuendo alla modularità delle applicazioni, tuttavia, citando un

articolo di google *"Closed shadow roots are not very useful. Some developers will see closed mode as an artificial security feature. But let's be clear, it's not a security feature. Closed mode simply prevents outside JS from drilling into an element's internal DOM."* [13]. Questo è evidenziato nell'articolo di Ankur Sundara [90], che mostra come nonostante le restrizioni stringenti implementate dai browser, sia possibile arrivare ad un interazione completa tra Shadow Host e Shadow DOM. Nel caso citato in precedenza, ad esempio, la funzione `window.find` è in grado di penetrare all'interno dello Shadow DOM, agendo in modo analogo alla ricerca testuale del browser (`CTRL+F`) su una pagina web. Questa funzione selezionerà la prima corrispondenza del testo, se presente. In aggiunta, è possibile estendere la selezione al massimo utilizzando il comando di Javascript:

```
document.execCommand("SelectAll")
```

che cattura tutto il testo, incluso quello cui potremmo non essere a conoscenza. Da qui, è possibile ottenere il contenuto del testo selezionato all'interno del Shadow DOM richiamando la funzione `window.getSelection`. Questa metodologia è efficace per esfiltrare il testo ed ottenere quindi un leak di informazioni. La leakage di informazioni, in questo contesto, si riferisce al rischio potenziale che dati critici e sensibili possano essere accessibili a terze parti non autorizzate attraverso il Shadow DOM.

Per ottenere in aggiunta un riferimento diretto ad un elemento all'interno dello Shadow DOM, in modo da poterlo manipolare completamente, su Firefox si può utilizzare la funzione `getSelection` che restituisce un oggetto `Selection`, dove `anchorElement` rappresenta un riferimento all'elemento all'interno dello Shadow DOM.

Tuttavia, Chromium implementa un'efficace misura di sicurezza tramite il sandboxing, e blocca il nodo restituito dall'`anchorNode`. E' possibile interagire comunque con lo Shadow DOM se all'interno dello stesso è possibile iniettare HTML o JS, ad esempio sfruttando l'attributo `contenteditable` su determinati elementi. Questo attributo HTML, ormai deprecato e poco utilizzato, dichiara la possibilità di modificare il contenuto dell'elemento da parte dell'utente. Attraverso l'uso del comando `window.find` e dell'API `document.execCommand`, è possibile interagire con un elemento `contenteditable` e ottenere un'iniezione di HTML. il comando `document.execCommand` accetta come parametro il valore `insertHTML`, che inserisce dell'HTML in sostituzione all'elemento. E' quindi possibile iniettare un tag, che quando viene caricato esegue del codice Javascript, che riesce quindi a interagire con lo Shadow DOM, esattamente come in un tipico attacco XSS, un esempio di payload è:

```
<svg/onload=console.log(this.parentElement.outerHTML)>
```

In aggiunta, è possibile dichiarare `contenteditable` su qualsiasi elemento in Chromium applicando una proprietà CSS ormai deprecata:

`-webkit-user-modify: read-write‘`

Ciò consente di elevare un'iniezione di CSS in un'iniezione di HTML, aggiungendo la proprietà CSS a un elemento e successivamente utilizzando il comando `insertHTML` come mostrato in precedenza.

Capitolo 6

Capitolo 7

Conclusioni

Nel corso di questa tesi, abbiamo esplorato in dettaglio le complesse sfide della sicurezza nei browser moderni, concentrandoci su diversi aspetti critici. Attraverso ciascun capitolo, abbiamo delineato vulnerabilità specifiche e le relative contromisure, mettendo in luce tanti aspetti fondamentali della sicurezza web.

Innanzitutto, abbiamo sottolineato l'importanza cruciale dell'adozione di HTTPS per garantire la crittografia delle comunicazioni web, evidenziando la necessità di una gestione attenta dei certificati, delle versioni SSL/TLS e la prevenzione del downgrade HTTPS. La gestione delle sessioni attraverso i cookie è stata esaminata in dettaglio, con un focus sugli attributi di sicurezza come `SameSite`, `Secure` e `HttpOnly`, la definizione accurata delle expiration policies, e l'utilizzo di pratiche come l'uso di prefixes. Ci siamo addentrati nell'architettura di politiche fondamentali come la Same-Origin Policy (SOP) e il Cross-Origin Resource Sharing (CORS). Mentre la SOP stabilisce confini di sicurezza, CORS offre un meccanismo controllato per la condivisione di risorse tra origini diverse. Tuttavia, vulnerabilità come JSONP, Origin Forgery e problemi legati a Cache HTTP sottolineano l'importanza di implementazioni rigorose. L'esecuzione sicura di codice nei browser è stata affrontata con un'analisi approfondita dell'architettura di base, esplorando vulnerabilità come buffer overflow, Use-After-Free e attacchi Javascript. Infine, l'isolamento della DOM attraverso tecnologie come Shadow DOM e IFRAME è stato esaminato, per mostrare vulnerabilità come Clickjacking e i rischi connessi a implementazioni errate. Header come `X-Frame-Options` e `Content-Security-Policy` sono emersi come gli strumenti utilizzati nella gestione di tali minacce.

In conclusione, per gestire con successo le sfide di sicurezza nei browser moderni, è fondamentale avere un approccio in costante aggiornamento. La profonda comprensione delle vulnerabilità, e l'implementazione di adeguate misure di mitigazione,

permettono di garantire un ambiente web più sicuro. Tramite questa strategia si fa in modo di rafforzare le linee di difesa principali contro le mutevoli minacce affrontate. La tesi spiega in maniera chiara e ben informata le intricazioni della sicurezza informatica in un panorama web in costante evoluzione.

Bibliografia

[1] Charles Reis Adam Barth Collin Jackson e Google Chrome Team. *The Security Architecture of the Chromium Browser*. URL: http://css.csail.mit.edu/6.858/2018/readings/chromium.pdf.

[2] *Advanced topics - Using data tainting for security*. Ago. 2002. URL: https://web.archive.org/web/20020808153106/http://wp.netscape.com:80/eng/mozilla/3.0/handbook/javascript/advtopic.htm#1009533.

[3] Christopher Allen e Tim Dierks. *The TLS Protocol Version 1.0*. RFC 2246. Gen. 1999. DOI: 10.17487/RFC2246. URL: https://www.rfc-editor.org/info/rfc2246.

[4] Apple. *About Security Update 2014-005*. Ott. 2014. URL: https://support.apple.com/en-us/HT203107.

[5] Apple. *About Security Update 2015-002*. Ott. 2014. URL: https://support.apple.com/en-us/103325.

[6] Apple. *About the security content of OS X Yosemite v10.10.4 and Security Update 2015-005*. Giu. 2015. URL: https://support.apple.com/en-us/HT204942.

[7] apple. *Apple security releases*. URL: https://support.apple.com/en-us/HT201222.

[8] Aidil Arief. *XSS Bypass sandbox="allow-same-origin" policy in IFRAME using the Latest version of Firefox Browser*. Ago. 2023. URL: https://www.secrash.com/2023/07/xss-bypass-sandbox-policy-in-iframe-using-latest-version-of-firefox-browser.html.

[9] Robert Auger. *The Cross-Site Request Forgery (CSRF/XSRF) FAQ*. Apr. 2008. URL: http://www.cgisecurity.com/csrf-faq.html.

[10] Richard Barnes et al. *Deprecating Secure Sockets Layer Version 3.0*. RFC 7568. Giu. 2015. DOI: 10.17487/RFC7568. URL: https://www.rfc-editor.org/info/rfc7568.

[11] Adam Barth. *HTTP State Management Mechanism*. RFC 6265. Apr. 2011. DOI: 10.17487/RFC6265. URL: https://www.rfc-editor.org/info/rfc6265.

[12] Adam Barth. *The Web Origin Concept*. RFC 6454. Dic. 2011. DOI: 10.17487/RFC6454. URL: https://www.rfc-editor.org/info/rfc6454.

[13] Eric Bidelman. *Shadow DOM v1 - Self-Contained Web Components*. Ago. 2016. URL: https://web.dev/articles/shadowdom-v1.

[14] Steven Bingler, Mike West e John Wilander. *Cookies: HTTP State Management Mechanism*. Internet-Draft. Work in Progress. Nov. 2023. URL: https://datatracker.ietf.org/doc/draft-ietf-httpbis-rfc6265bis/13/.

[15] Sharon Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. Mag. 2008. DOI: 10.17487/RFC5280. URL: https://www.rfc-editor.org/info/rfc5280.

[16] Wouter Bokslag. "The problem of popular primes: Logjam". In: *CoRR* abs/1602.02396 (2016). arXiv: 1602.02396. URL: http://arxiv.org/abs/1602.02396.

[17] Will Boucher. *SameSite: Hax – Exploiting CSRF With The Default SameSite Policy*. Feb. 2022. URL: https://pulsesecurity.co.nz/articles/samesite-lax-csrf.

[18] caniuse.com. *headers HTTP header: Set-Cookie: Cookie prefixes*. URL: https://caniuse.com/mdn-http_headers_set-cookie_cookie_prefixes.

[19] Jianjun Chen et al. "We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, ago. 2018, pp. 1079–1093. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/chen-jianjun.

[20] Ari Chivukula. *Cookie Expires and Max-Age attributes now have upper limit*. Gen. 2023. URL: https://developer.chrome.com/blog/cookie-max-age-expires/.

[21] Google Chrome. *Feature: Cookies default to SameSite=Lax*. Lug. 2020. URL: https://chromestatus.com/feature/5088147346030592.

[22] chromium.com. *Chromium issue tracker*. URL: https://bugs.chromium.org/p/chromium/issues/list?q=Type%3DBug-Security%20status%3DFixed&can=1.

[23] *Content Security Policy Level 3 - frame-ancestors*. Ott. 2023. URL: https://w3c.github.io/webappsec-csp/#directive-frame-ancestors.

[24] *Content Security Policy Level 3 - Relation to X-Frame-Options*. Ott. 2023. URL: https://w3c.github.io/webappsec-csp/#frame-ancestors-and-frame-options.

[25] *CVE-2011-2382*. Giu. 2011. URL: https://nvd.nist.gov/vuln/detail/CVE-2011-2382.

[26] *CVE-2011-3389*. Set. 2011. URL: https://nvd.nist.gov/vuln/detail/CVE-2011-3389.

[27] *CVE-2014-0160*. Lug. 2014. URL: https://nvd.nist.gov/vuln/detail/cve-2014-0160.

[28] *CVE-2014-3566*. Ott. 2014. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-3566.

[29] *CVE-2015-0807*. Apr. 2015. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-0807.

[30] *CVE-2015-0810*. Apr. 2015. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-0810.

[31] *CVE-2015-1067*. Mar. 2015. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-1067.

[32] *CVE-2015-3658*. Lug. 2015. URL: https://nvd.nist.gov/vuln/detail/CVE-2015-3658.

[33] *cve-2015-4000*. Mag. 2015. URL: https://nvd.nist.gov/vuln/detail/cve-2015-4000.

[34] *CVE-2019-5861*. 2019 11. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-5861.

[35] *CVE-2021-30528*. Giu. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-30528.

[36] *CVE-2022-1529*. Dic. 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-1529.

[37] *CVE-2023-36719*. Nov. 2023. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-36719.

[38] cve.mitre.org. *Search Results for: chrome sandbox escape*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=chrome+sandbox+escape.

[39] cve.mitre.org. *Search Results for: Edge sandbox escape*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Edge+sandbox+escape.

[40] cve.mitre.org. *Search Results for: firefox sandbox escape*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=firefox+sandbox+escape.

[41] Tim Dierks e Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346. Apr. 2006. DOI: 10.17487/RFC4346. URL: https://www.rfc-editor.org/info/rfc4346.

[42] *Fetch Living Standard - CORS protocol*. Nov. 2023. URL: https://fetch.spec.whatwg.org/#cors-protocol.

[43] Roy T. Fielding, Mark Nottingham e Julian Reschke. *HTTP Semantics*. RFC 9110. Giu. 2022. DOI: 10.17487/RFC9110. URL: https://www.rfc-editor.org/info/rfc9110.

[44] J.R. Prins (CEO Fox-IT). *DigiNotar Certificate Authority breach "Operation Black Tulip"*. Set. 2011. DOI: 10.13140/2.1.2456.7364. URL: https://www.bitsoffreedom.nl/wp-content/uploads/rapport-fox-it-operation-black-tulip-v1-0.pdf.

[45] Alan O. Freier, Philip Karlton e Paul C. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101. Ago. 2011. DOI: 10.17487/RFC6101. URL: https://www.rfc-editor.org/info/rfc6101.

[46] Google. *Chrome 45 release*. Set. 2015. URL: https://chromereleases.googleblog.com/2015/09/stable-channel-update.html.

[47] Google. *Chrome 84 release notes*. Dic. 2018. URL: https://developer.chrome.com/blog/chrome-84-deps-rems/.

[48] Google. *Chrome HSTS Preload List*. URL: https://www.chromium.org/hsts/#preloaded-hsts-sites.

[49] Google. *Code Reviews: Add minimum TLS version control to about:flags and Finch gate it*. Gen. 2015. URL: https://codereview.chromium.org/693963003.

[50] Google. *CRLSets*. URL: https://www.chromium.org/Home/chromium-security/crlsets/.

[51] A. Grosskurth e M.W. Godfrey. *A reference architecture for Web browsers*. 2005. DOI: 10.1109/ICSM.2005.13.

[52] Jeremiah Grossman. *Cross-Site Tracing (XST) The new techniques and emergin threats to bypass current web security measures using TRACE and XSS*. Gen. 2003.

[53] Jeff Hodges, Collin Jackson e Adam Barth. *HTTP Strict Transport Security (HSTS)*. RFC 6797. Nov. 2012. DOI: 10.17487/RFC6797. URL: https://www.rfc-editor.org/info/rfc6797.

[54] Bob Ippolito. *Remote JSON - JSONP*. Dic. 2005. URL: https://web.archive.org/web/20091204053053/http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/.

[55] Soheil Khodayari e Giancarlo Pellegrino. *The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies*. 2022. DOI: 10.1109/SP46214.2022.9833637.

[56] Adam Langley. *Safari HSTS Preload List*. Dic. 2013. URL: https://twitter.com/agl__/status/414112266938617856.

[57] Hossein Lotfi. *But You Told Me You Were Safe: Attacking the Mozilla Firefox Sandbox (Part 2)*. Ago. 2022. URL: https://www.zerodayinitiative.com/blog/2022/8/23/but-you-told-me-you-were-safe-attacking-the-mozilla-firefox-renderer-part-2.

[58] Giulio Candreva Marco Bartoli. *Escaping the sandbox: A bug that speaks for itself*. Nov. 2023. URL: https://microsoftedge.github.io/edgevr/posts/Escaping-the-sandbox-A-bug-that-speaks-for-itself/.

[59] Jörg Schwenk Marcus Niemietz. *UI Redressing Attacks on Android Devices*. URL: https://media.blackhat.com/ad-12/Niemietz/bh-ad-12-androidmarcus_niemietz-WP.pdf.

[60] Microsoft. *Edge 84 release notes*. Lug. 2020. URL: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-relnote-archive-stable-channel.

[61] Microsoft. *Edge HSTS Preload List*. Giu. 2015. URL: https://blogs.windows.com/msedgedev/2015/06/09/http-strict-transport-security-comes-to-internet-explorer-11-on-windows-8-1-and-windows-7/.

[62] Microsoft. *Happy 10th birthday Cross-Site Scripting!* Dic. 2009. URL: https://learn.microsoft.com/en-ca/archive/blogs/dross/happy-10th-birthday-cross-site-scripting.

[63] Microsoft. *Microsoft Security bulletins*. URL: https://learn.microsoft.com/en-us/security-updates/securitybulletins/securitybulletins.

[64] Microsoft. *Release notes for Microsoft Edge Security Updates*. URL: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-relnotes-security.

[65] Microsoft. *SameSite cookie attribute: 2020 release*. Feb. 2020. URL: https://learn.microsoft.com/en-us/microsoftteams/platform/resources/samesite-cookie-update#samesite-cookie-attribute-2020-release.

[66] miTLS. *SMACK: State Machine AttaCKs*. URL: https://mitls.org/pages/attacks/SMACK.

[67] Man Yue Mo. *The fugitive in Java: Escaping to Java to escape the Chrome sandbox*. Set. 2021. URL: https://securitylab.github.com/research/chrome_sbx_java/.

[68] Kathleen Moriarty e Stephen Farrell. *Deprecating TLS 1.0 and TLS 1.1*. RFC 8996. Mar. 2021. DOI: 10.17487/RFC8996. URL: https://www.rfc-editor.org/info/rfc8996.

[69] Mozilla. *Changes to SameSite Cookie Behavior – A Call to Action for Web Developers*. Ago. 2020. URL: https://hacks.mozilla.org/2020/08/changes-to-samesite-cookie-behavior/.

[70] Mozilla. *Firefox 34 POODLE fix*. Ott. 2014. URL: https://blog.mozilla.org/security/2014/10/14/the-poodle-attack-and-the-end-of-ssl-3-0/.

[71] Mozilla. *Firefox 34 release notes*. Dic. 2014. URL: https://www.mozilla.org/en-US/firefox/34.0/releasenotes/.

[72] Mozilla. *Firefox 78 release notes*. Giu. 2020. URL: https://www.mozilla.org/en-US/firefox/78.0/releasenotes/.

[73] Mozilla. *Firefox HSTS Preload List*. URL: https://wiki.mozilla.org/Privacy/Features/HSTS_Preload_List.

[74] Mozilla. *Firefox Logjam fix*. Lug. 2015. URL: https://www.mozilla.org/en-US/security/advisories/mfsa2015-70/.

[75] Mozilla. *Security Advisories for Firefox*. URL: https://www.mozilla.org/en-US/security/known-vulnerabilities/firefox/.

[76] Mozilla. *Set-Cookie ¿ Compatibility notes*. Feb. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#compatibility_notes.

[77] mozilla. *Revoking Intermediate Certificates: Introducing OneCRL*. Mar. 2015. URL: https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/.

[78] mozilla. *SSL 0.2 PROTOCOL SPECIFICATION*. Feb. 1995. URL: https://www-archive.mozilla.org/projects/security/pki/nss/ssl/draft02.html.

[79] Tali Garsiel Paul Irish. *How browsers work*. Ago. 2011. URL: https://web.dev/articles/howbrowserswork?source=post_page-----840c079e1e0a.

[80] Tim Polk e Sean Turner. *Prohibiting Secure Sockets Layer (SSL) Version 2.0*. RFC 6176. Mar. 2011. DOI: 10.17487/RFC6176. URL: https://www.rfc-editor.org/info/rfc6176.

[81] Qualys. *About Security Update 2014-005*. Feb. 2014. URL: https://qualys.my.site.com/discussions/s/question/0D52L00004TnuLRSAZ/apple-finally-releases-patch-for-beast.

[82] Eric Rescorla. *HTTP Over TLS*. RFC 2818. Mag. 2000. DOI: 10.17487/RFC2818. URL: https://www.rfc-editor.org/info/rfc2818.

[83] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Ago. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[84] Eric Rescorla e Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Ago. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[85] David Ross e Tobias Gondrom. *HTTP Header Field X-Frame-Options*. RFC 7034. Ott. 2013. DOI: 10.17487/RFC7034. URL: https://www.rfc-editor.org/info/rfc7034.

[86] Gustav Rydstedt et al. "Busting frame busting a study of clickjacking vulnerabilities on popular sites". In: *Web 2.0 Security and Privacy*. IEEE. 2010.

[87] *Same Origin Policy*. Gen. 2010. URL: https://www.w3.org/Security/wiki/Same_Origin_Policy.

[88] Stefan Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. Giu. 2013. DOI: 10.17487/RFC6960. URL: https://www.rfc-editor.org/info/rfc6960.

[89] statcounter.com. *Browser Market Share Worldwide*. Nov. 2023. URL: https://gs.statcounter.com/browser-market-share/all/worldwide/2023.

[90] Ankur Sundara. *The Closed Shadow DOM*. Mag. 2022. URL: https://blog.ankursundara.com/shadow-dom/.

[91] Moxie Marlinspike - Blackhat talk. *New Tricks For Defeating SSL In Practice*. 2009. URL: https://blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf.

[92] Microsoft Edge Team. *Microsoft Edge and Chromium Open Source: Our Intent*. Dic. 2018. URL: https://github.com/MicrosoftEdge/MSEdge/blob/7d69268e85e198cee1c2b452d888ac5b9e5995ca/README.md.

[93] Luke Valenta et al. *Factoring as a Service*. A cura di Jens Grossklags e Bart Preneel. Berlin, Heidelberg, 2017.

[94] Rosario Valotta. *Cookiejacking*. Mag. 2011. URL: https://archive.conference.hitb.org/hitbsecconf2011ams/materials/D2T2%20-%20Rosario%20Valotta%20-%20Cookie%20Jacking.pdf.

[95] webcomponents.org. *Webcomponents Specifications*. URL: https://www.webcomponents.org/specs.

[96] Mike West e Mark Goodwin. *Same-Site Cookies*. Internet-Draft. Work in Progress. Giu. 2016. URL: https://datatracker.ietf.org/doc/draft-ietf-httpbis-cookie-same-site/00/.

[97] whatwg. *DOM Living Standard*. Ott. 2023. URL: https://dom.spec.whatwg.org/.

[98] whatwg. *HTML Living Standard - The iframe element*. Nov. 2023. URL: https://html.spec.whatwg.org/multipage/iframe-embed-object.html.

Ringraziamenti

Questa avventura non sarebbe stata possibile senza il contributo delle persone che mi sono state accanto, e per questo voglio ringraziarle di cuore. Credo davvero che senza di loro sicuramente adesso non sarei qui, sono state tutte, a modo loro, fondamentali per arrivare alla fine di questo percorso.

Ringrazio mia madre, Monica, e mio padre, Cristiano, per avermi dato la possibilità di partire a Bologna. Senza il vostro appoggio non avrei potuto iniziare questa esperienza fantastica, e grazie ai vostri insegnamenti e al vostro supporto sono riuscito ad affrontare al meglio tutte le sfide che si sono presentate in questi ultimi 3 anni.

Ringrazio mia sorella, Beatrice, per essere gentile con me a prescindere da tutto, anche in quelle situazioni in cui puoi tranquillamente fare finta di nulla, e anche se non sono sempre stato presente nei tuoi confronti. Mi hai mostrato che nonostante le differenze, siamo sempre fratelli, e per questo ti ringrazio.

Ringrazio i miei nonni, Marisa e Antonello, perché mi pensate ogni giorno. Con i vostri piccoli e grandi gesti quotidiani, mi fate emozionare ogni volta. Tutte le volte che ci siamo sentiti, che mi avete fatto spedire le cose da mangiare (che senza seadas non si può stare), che mi avete fatto dei regali, o anche solo che ho sentito i vostri pensieri rivolti verso di me, sono riuscito a sentire un amore profondo nei miei confronti, che solo voi mi potete dare. Non potrò mai ripagarvi per tutto quello che fate, per il vostro impegno e la vostra pazienza instancabile. Spero almeno di rendervi fieri di me.

Ringrazio i miei nonni, Gesuino e Ignazia, per essermi stati vicini durante questo viaggio, e per avermi insegnato tanto. Nonna, mi hai sempre dato tanto amore e cura, e per questo te ne sono grato. Nonno, so che non puoi leggere questo messaggio, ma ti porterò sempre nel mio cuore. Nei momenti difficili ricordo sempre il tuo sorriso, mi fa ricordare che le cose belle nella vita stanno nelle piccole cose. Un giorno spero di diventare un nonno che fa ridere i suoi nipoti con lo stesso sguardo con cui mi hai fatto divertire anche tu, gli racconterò che pescare i pesci piccoli è

più difficile di pescare quelli grandi, e spero di strappargli un sorriso. Mi manchi tanto.

Ringrazio tutti i miei zii, Alessandro, Olivia, Gigi, Carlo, Debora, Luca e Stefania, e i miei cugini, Andrea, Vittoria, Matilde, Alice, Marco, Giulia e Sofia, per essere stati presenti e avermi dato supporto in questo percorso di vita, ho notato tutti i gesti che ognuno di voi, singolarmente, ha rivolto verso di me, e non posso fare a meno di essere grato del trattamento che mi riservate ogni volta. Vicky, a te un ringraziamento particolare, mi hai aperto tanti orizzonti e sei stata di enorme aiuto in questo percorso, con l'enorme pazienza che hai avuto nei miei confronti e tutto l'aiuto che mi hai dato, sono davvero contento di averti come cugina.

Ringrazio poi i miei zii, Patrizia e Antonluca. Mi avete sempre accolto come un figlio, ogni volta che vi parlo mi rendo conto che tenete a me in un modo che non si può descrivere a parole. Mi pensate anche quando non ci sentiamo, mi ascoltate e supportate incondizionatamente, mettendo sempre in primo piano i miei interessi e bisogni. Spero che questo traguardo possa dare un po' di gioia anche a voi, vi voglio bene.

Ringrazio Stefania, Massimo, Nella e Giorgio, per avermi sempre trattato come un nipote. Avete sempre tenuto al mio benessere, non avete mai mancato di dimostrarmelo. Mi avete insegnato tantissimo, dandomi ottimi consigli, da quelli sulla barba a quelli sulla carriera, prendendo sempre a cuore i miei problemi. Ogni volta che raggiungo un traguardo, vedo nei vostri occhi una gioia più grande di quella che potrebbe darvi raggiungere il traguardo voi stessi, non saprei come altro descrivere il bene che mi volete. So che credete in me sempre, e mi date una fiducia incondizionata, grazie a voi mi sento accolto. Vi ringrazio di cuore.

In questa avventura, da lontano e da vicino, i miei amici hanno giocato un ruolo fondamentale. Luca, Matteo, Davide, Gabriele, siete degli amici fantastici. Mi ascoltate sempre, dandomi ottimi consigli, e mettete in primo piano i miei sentimenti, come solo voi sapete fare. Ci siamo divertiti tanto insieme, tra meme (a volte troppi raga dovremmo darci una calmata), partite online e feste di ogni genere, grazie a voi ho avuto la carica di affrontare questi tre anni con serenità, grazie a voi sono riuscito ad affrontare i momenti di tristezza e sconforto, sono veramente grato di sentirvi ogni volta. Luca, ormai mi sei accanto da più di un decennio. Mi conosci meglio di qualsiasi altra persona, perché mi sei stato sempre vicino, nei momenti più difficili, e nei momenti più felici. Ci conosciamo da tanto che vorrei aggiungere un meme, ma ne abbiamo veramente troppi e non so decidermi, quindi "ciabatta". Insieme abbiamo riso, pianto, litigato, abbiamo provato tutte le emozioni che si possono provare. Sei la roccia su cui so di poter sempre contare, la persona che so sempre di poter chiamare quando tutto va veramente storto, la persona che gioisce sempre

accanto a me quando ho una conquista, e che mi tira su quando sono completamente a terra. Grazie a te so che posso affrontare la vita con sicurezza. Sei la mia spalla, amico mio, sei un fratello. Con te riesco ad essere veramente felice, grazie. Cance, tu mi hai dato ascolto, mi hai guardato davvero dentro, e mi sei stato vicino anche nei momenti più bui. Mi hai sempre dato ottimi consigli (tranne con la musica, ascolti roba strana a volte amico mio), hai sempre cercato di aiutarmi e prepararmi ad ogni evenienza, mi trasmetti sempre tanto affetto. So di poter contare su di te, e non vedo l'ora di poter festeggiare qualcosa di tuo, importante, per renderti tutto l'aiuto che mi dai. Davi, abbiamo lo stesso cognome ma non abbiamo legami di parentela, nonostante questo mi tratti come se fossi parte della tua famiglia. Vedo quanto cuore metti nelle cose che fai, e l'attenzione che dai ai tuoi amici. Ci tieni incondizionatamente, nel vero senso del termine, tu vuoi il bene per noi (anche quando facciamo roba assurda tipo portarti in dono un anguria). Ho sempre ammirato questo tuo lato, sei un grande amico, e ti auguro il meglio per la tua avventura in Spagna, spero di riuscire ad esserti d'aiuto esattamente come tu hai fatto con me fino ad ora, grazie. Gabri, ho tanti bei ricordi con te. Ci siamo fatti un sacco di risate insieme, e abbiamo condiviso tantissimo, tra gusti musicali e segreti, ogni volta che ci vediamo ci divertiamo come se fosse la prima. Spero che la tua avventura a "M come Brescia" si concluda nel migliore dei modi, grazie mille del supporto che mi dai.

Susanna, l'ultimo anno e mezzo insieme è stato ricco di emozioni, nessuno mi è stato vicino come te. La parola grazie, nei tuoi confronti, non esprime minimamente quello che provo. Sei una ragazza fantastica, vivi la vita con il cuore, e affronti le avversità con una passione che non ho mai visto. Ogni volta che stiamo assieme mi sento come fosse la prima, non mi stanco mai della tua sfacciataggine, del tuo modo di indorare la pillola, del tuo modo di fare alla "sono più importante io", del tuo modo di mettere in luce le persone che hai intorno, della tua rabbia, della tua gioia, dei tuoi dubbi e della tua ansia, della tua convinzione nel fare le cose più assurde, delle tue foto senza senso, della tua attenzione ai dettagli, del tuo mal di testa, dei tuoi scatti di energia prima di dormire, del tuo dentino, del tuo sorriso, delle tue mosse di danza buffe, dei tuoi occhi a mandorla, dei tuoi racconti sconclusionati, di quando canti e mi fai emozionare, dei tuoi pianti, delle tue risate, dei tuoi sorrisi. La verità è che ogni secondo con te è speciale, hai fatto brillare tutto il tempo passato insieme, non posso stancarmi di niente, riesco solo a pensare a quanto siamo stati bene, a quanto mi hai aiutato, e a quanto mi hai pensato ogni momento. Mi sei stata vicino nonostante tutto, ma davvero tutto, e mi hai permesso di affrontare l'enorme impegno degli ultimi tempi con leggerezza, di dimenticare i problemi. Grazie a te ho scoperto cosa significa tenere ad una persona, ho scoperto cosa significa aiutare dal profondo. Sei la mia medicina, con te mi sento a casa.

Ringrazio poi Marco, sai tutte le esperienze che abbiamo vissuto insieme, quante risate, quante ore abbiamo passato a parlare di anime, teorie assurde, videogiochi e cose completamente senza senso. In tutti questi momenti mi hai insegnato tanto, mi sei stato d'aiuto, grazie a te ho scoperto come lasciar perdere le cose che non mi interessano, e ho potuto godermi completamente tante esperienze. Non sarei la persona che sono adesso se non ti avessi conosciuto, credo fermamente che questa mia esperienza universitaria sarebbe andata in maniera profondamente diversa senza quello che ci siamo detti nel tempo, ti ringrazio. L'avventura che ormai intraprendi da un po' spero ti porti a trovare il tuo posto felice, ti auguro sempre il meglio, ogni volta che ci vediamo mi strappi un sorriso.

Riccardo, forse dovrei dire rekke, ci conosciamo da tanto, e in tutti questi anni abbiamo coltivato un rapporto di enorme fiducia. Ho sempre ammirato la tua perseveranza e dedizione alle cose a cui tieni, hai sempre le idee chiare e quando ti metti in testa un obbiettivo lavori fino a quando non lo raggiungi. Questo tuo modo di fare mi ha motivato tanto durante questo percorso, permettendomi di arrivare alla fine, e per questo ti ringrazio tanto. Ho sempre contato su di te, e tu non mi hai mai chiesto niente. Hai fatto un sacco per me, e sei sempre stato pronto ad aiutarmi, a prescindere da tutto quanto. Sei un vero amico. Grazie ancora, ti meriti tutto il bene che so darti.

Infine devo assolutamente dedicare i miei ringraziamenti anche a tutte le persone che a Bologna mi hanno dato una mano, che mi sono state vicine.

Ringrazio Francesco, o meglio Apo, abbiamo passato un anno spettacolare in casa, completamente in sintonia, non avrei mai detto che sarei riuscito a legare tanto in cosi poco tempo. In te ho trovato un grande amico, una persona con cui condividere gioie e dolori, sei stata una spalla (un po' alta ma ce la facciamo andare bene) su cui ho potuto contare durante questo percorso. Purtroppo per il momento le nostre strade si sono divise, ma spero di poterti incontrare in futuro. Ti ringrazio per tutto, mi hai insegnato a godermi il presente, sei riuscito a rendere leggero tutto il tempo passato insieme. Ti auguro il meglio, grazie ancora.

Desidero ringraziare Gabriele, che ormai per me si chiama Geno. Da quando sono arrivato nella tua città non hai fatto altro che supportarmi e sopportarmi, mi hai portato con te quando mi sentivo solo, e senza fare domande hai sempre cercato di risolvere i miei problemi e le mie paranoie, anche quando mi conoscevi a malapena. Ho solo bei ricordi con te, sei una persona di cuore e provo profonda tristezza a sapere che non posso più vederti tutti i giorni perché abitiamo lontani. Ti ammiro tanto per quello che fai con tutti, mi hai insegnato il significato di essere buono con il mondo. Grazie.

Daniele, a te devo un grande ringraziamento, mi hai dato ospitalità quando più ne

ho avuto bisogno, mi sei stato accanto quando mi sentivo abbandonato, e hai sempre pensato a me e a come aiutarmi. Il tempo a Bologna non sarebbe stato lo stesso senza la tua presenza, le tue battute e il nostro costante tentativo di infastidirci a vicenda. Ci siamo divertiti tanto insieme, ti ringrazio infinitamente per la presenza positiva che mi hai donato in questo cammino. Ti auguro di prendere questa tesi di traverso, sai che intendo.

Ringrazio il mio capo Massimiliano e i miei colleghi Stefano e Francesco, che in DigitalDefense mi hanno accolto e trattato subito come un membro del team. Ho imparato veramente tanto, grazie alla possibilità che mi avete dato di lavorare con voi. Grazie alla vostra disponibilità e generosità sono sicuramente riuscito ad affrontare al meglio questo ultimo anno di università. Mi avete permesso di concludere in tempo questa avventura, e penso che senza di voi sarebbe andata diversamente, per questo vi ringrazio.

Infine ringrazio tutto il gruppo Ranzani, Gian, Volpe, Luca, Yonas, Erik, Simone, Fabio, Andrea, Andreea, Francesca, Federica, Paolo e Betto. Grazie per il supporto che mi è stato dato da tutti, per le sessioni di studio, per tutte le informazioni che sono state condivise, per le risate fatte durante questi anni. Tutto ciò che abbiamo condiviso in questo periodo è stato fondamentale per darmi la motivazione di arrivare in fondo a questo percorso, grazie ancora.