HTTP Integrity: A Lite and Secure Web against World Wide Woes

Taehwan Choi
Department of Computer Sciences,
University of Texas at Austin

Mohamed G. Gouda
Department of Computer Sciences,
University of Texas at Austin
National Science Foundation

Abstract

While there is no guarantee of HTTP page integrity, this issue is left unaddressed in discussions of web security. Though HTTPS can be used to solve the HTTP page integrity problem, HTTPS is shunned by web communities due to the performance overheads caused by TLS. Worse yet, HTTPS inherently breaks the distributed nature of the web by disallowing caching. The end-to-end security guarantee of HTTPS only allows web contents served by origin web servers, not caching proxies or Content Delivery Networks (CDN). Unsurprisingly, HTTPS is overkill for many applications and is avoided by many websites. Thus, webpages are completely open to attacks against HTTP page integrity. Based on these observations, we have designed a lite protocol for secure web, HTTP Integrity (HTTPI). HTTPI relies on HTTPS to share session keys and use them for keyed-hashing HTTP pages. We show that HTTPI can be reliably used for many applications, since many web attacks target integrity rather than confidentiality. In order to avoid breaking the caching mechanism of the web, we decouple HTTP headers and contents for keyed-hashing. Web servers can cache or precompute contents hashing for static contents and many studies show that dynamic contents can be cached as well. Therefore, the performance degradation caused by HTTPI can go unnoticed by users.

1 Introduction

As the World Wide Web focuses on scalability and performance rather than security, it suffers widely from classes of attacks including server impersonation, message modification, cookie theft and cookie injection. Contributing to these problems, wireless networks proliferate, and any attacker can easily eavesdrop and modify traffic from web clients in his or her proximity areas. HTTPS can stop these problems by providing confidentiality and integrity, but HTTPS is widely shunned by many websites. The performance overheads caused by TLS are not the only reason for this contradictory situation. It also results from the end-to-end security of HTTPS, which prevents websites from distributing the content over the Internet to reduce bandwidth and latency. These "World Wide Woes" will continue unless we take proactive actions against them.

The recent trends of web attacks show that web attacks are more evolved and advanced than ever before. Cookie eavesdropping such as Side Jacking[15] occurred in 2007 and showed that authentication cookies can be eavesdropped and replayed while a user is using HTTP after the user is authenticated by HTTPS. More advanced cookie theft was found as Surf Jacking[14] in 2008. An attacker impersonates any website using ARP poisoning and redirects a user to any target website to steal cookies. In 2009, the first Man-in-the-Middle (MITM) attack in the web was introduced as SSLStrip[32]. Most HTTPS-enabled websites allow a user to initiate an HTTPS session with HTTP. If a user types an URL without `https`, `http` is used for the default scheme for a website. Then, the website redirects the user to `https`. SSLStrip makes use of this convention and establishes an HTTP connection between a user's browser and an attacker, and establishes an HTTPS connection between an attacker and a target website. Though SSLStrip shows how to use an HTTP-to-HTTPS redirection and attacks against HTTPS, this possibly points out the vulnerability against HTTP page integrity. Any attacker can modify HTTP pages and inject malicious scripts or steal session cookies unnoticed by a user even when the user browses a legitimate website[42]. As long as this HTTP integrity problem remains, it is doomed to be a future disaster.

Despite these complications, the available defense of HTTP is not beyond the Same-Origin Policy (SOP). In order to understand SOP, we need to consider SOP in the context of a model using both subjects and objects. SOP is access control policy for a browser. Subjects

are any HTML documents or loaded scripts and objects are DOM objects, Cookies[31], and HTML documents. Subjects can access objects if subjects satisfy all the matching rules with objects depending on the types of SOP. We define SOP and Cookie SOP as follows:

- SOP: Subject S can access object O only if $protocol_S = protocol_O$, $domain_S = domain_O$, and $port_S = port_O$
- Cookie SOP: Subject S can access cookie O only if $domain_S = domain_O$, $path_S = path_O$, and $protocol_S =$ "$https$" when $secure_O = true$

Each attribute has the different matching rule. The matching rule for the protocol and the port attribute follows equality. The protocol attribute of subjects should be equal to that of objects and the port attribute of subjects should be equal to that of objects. The matching rule for the domain attribute in SOP and Cookie SOP follows the *longest suffix rule*. The longest suffix of the domain attribute in objects should match that in subjects. Likewise, the matching rule for the path attribute follows the *longest prefix rule*. The longest prefix of the path attribute in objects should match that in subjects. SOP seems effective against unauthorized access to web objects, but it can be breached by an attacker due to the lack of integrity in HTTP. HTTP's dependence on SOP without integrity causes the following four architectural problems, which will be detailed in Section 5.

P1. Misbinding: An attacker can impersonate a web server by associating an attacker's IP address with a legitimate website's domain name or an attacker's MAC address with a legitimate IP address.

P2. Message modification: An attacker can modify HTTP headers[27, 28] or contents[40]. If an attacker injects a malicious script or a link to a malicious script in an HTTP page, the script is executed in the context of the HTTP page.

P3. Cookie injection and theft: An attacker can overwrite the cookies in a user's browser with the attacker's cookies[29]. An attacker can steal cookies from users by eavesdropping[15] or impersonation[14] and replay the cookies. These problems can be resolved by setting the secure attribute for cookies. However, a study shows that one third of 30 websites did not set the secure attribute for authentication cookies[5].

P4. HTTP-HTTPS interaction: The security of HTTPS can be broken by using the redirection from HTTP to HTTPS[32]. The HTTP-to-HTTPS redirection can be done by upgrading to TLS within HTTP/1.1[25] and using redirection messages like status code 301 (Moved Permanently), 303 (See Other), and 307 (Temporary Redirect). If an HTTPS page contains HTTP links, HTTP links can be used to break the security of HTTPS[5].

We will describe the security guarantees of HTTPI in Section 4. In Section 5, we will show that these guarantees can defend against many attacks including server impersonation, message modification, cookie injection and cookie theft.
This paper is organized as follows: Section 2 discusses why we should use HTTPI instead of HTTP, HTTPS, and HTTP Authentication in terms of server authentication, client authentication, integrity, and confidentiality. Section 3 presents our design of HTTPI focusing on HTTPI sessions, and cookies. Section 4 shows five security guarantees that HTTPI provides. In Section 5, we categorize and analyze web attacks and show that many web attacks are related to integrity, and attacks not defended by HTTPI also can not be defended by HTTPS. Section 6 describes our implementation in detail. Section 7 compares the performance of HTTPI in terms of throughput, and CPU time with HTTP and HTTPS, and discusses HTTPI performance in terms of bandwidth and latency. Section 8 reviews related work. Finally, Section 9 summarizes and concludes our work.

2 Motivation

In this section, we discuss why we should use HTTPI instead of HTTP, HTTPS, or HTTP Authentication by comparing them in terms of server authentication, client authentication, integrity and confidentiality. We summarize our analysis in Table 1.

2.1 Why Not HTTP?

HTTP[9] is a request and response protocol and if a web client sends an HTTP request to a web server, the web client receives an HTTP response from the web server. If each request and response is defined as a transaction, each transaction is independent and thus HTTP is a stateless protocol. In order to manage states among transactions, cookies[31] were proposed to manage states in web clients. Web servers send cookies, texts with names and values pairs, to web clients and set cookies to them and web clients send the cookies in subsequent requests to the same web server to notify the states of the web clients. Due to these facts, HTTP is scalable and contributes to the success of the current web architecture. In spite of the success, the integrity problems of HTTP are widely under appreciated. HTTP has no server and

	Server Auth	Client Auth	Integrity	Confidentiality
HTTP				
HTTPS	✓		✓	✓
HTTPS with Password	✓	✓	✓	✓
HTTP Authentication		✓	✓	
HTTPI	✓		✓	
HTTPI with Password	✓	✓	✓	

Table 1: Comparisons between HTTPI, HTTP, HTTPS, and HTTP Authentication

client authentication. HTTP messages can be modified and eavesdropped in-transit.

2.2 Why Not HTTPS?

HTTPS[41] is TLS[6] over HTTP, and provides confidentiality, integrity, and server authentication. HTTPS provides the strongest security guarantee among HTTP, HTTPS, HTTP Authentication and HTTPI. If HTTPS is used between a web client and a web server, HTTPS pages can not be modified and eavesdropped in-transit and many web attacks can be mitigated. Even with these strong security improvements, HTTPS has not replaced HTTP completely. HTTPS is used minimally when it is required for authentication, online shopping and online banking. Currently, HTTPS is shunned by many websites due to primarily performance degradation. If HTTPS affects only the performance, HTTPS will be adopted completely as computers and networks are much faster than now in the future. The more reasonable answer to the phenomena must be more fundamental in web architecture, and HTTPS does break the current web architecture. HTTP is a distributed information system depending on many intermediary entities between a web client and a web server such as proxies and gateways, and HTTP pages are cached in caching proxies and replicated in gateways as Content Delivery Networks (CDNs)[12]. On the other hand, HTTPS provides the end-to-end security, and any HTTPS pages can not be cached by proxies and gateways, but should be served by origin web servers. Furthermore, HTTPS pages can not be incrementally rendered and HTTPS pages look more sluggish to users. These facts increase bandwidth and latency in the Internet. In many web applications such as social networking websites like Facebook and news websites like the New York Times, HTTPS is overkill since confidentiality is not as critical as other websites.

2.3 Why Not HTTP Authentication?

HTTP Authentication[10] is access authentication protocol, and provides client authentication based on a password system. Basic Authentication is proposed in HTTP/1.0[3] and user names and passwords are sent in clear text, and Digest Authentication[11] is proposed to remedy the problem by sending user names and passwords by hashing. Since HTTP Authentication relies on a password system for client authentication, HTTP Authentication has inherent problems from the password system. HTTP Authentication does not provide server authentication, and HTTP Authentication is vulnerable to Man-in-the-middle (MITM) attacks. Though HTTP Authentication provides integrity, many features are optional and never used in practice. We found that Apache 2.2.11 web server did not implement the optional content integrity feature for Digest Authentication. HTTP header fields are not protected by digest, but some selective header fields such as the request-uri value and the method value from the Request-Line are hashed. Thus, HTTP Authentication does not provide complete integrity. Most of all, HTTP Authentication is shunned by many websites and Form-based Authentication is readily adopted due to usability[45]. HTTP Authentication invokes a password dialog box dependent on browsers, and does not integrate with web applications. Though HTTP Authentication can be better than Form-based Authentication in terms of security, HTTP Authentication confuses users and is avoided by many users.

2.4 Why HTTPI?

Our HTTPI provides integrity and server authentication by keyed-hashing HTTP header fields and contents. Our HTTPI does not provide client authentication like HTTPS, but HTTPS and our HTTPI can support client authentication by adopting a password system. The most distinctive feature that our HTTPI does not support is confidentiality compared to HTTPS. The rational behind this is two fold. First, encrypted pages can not be cached and break the web architecture: HTTPS uses different keys for different sessions, a page in one session can not be the same page in another session if pages are encrypted. Second, HTTPS security is overkill for many applications since confidentiality is not required for every application. Therefore, we sacrifice confidentiality for two benefits. First, our HTTPI pages can be cached

like HTTP pages and our HTTPI does not break the current web architecture. Second, our HTTPI performs better than HTTPS without encryption. We show that many web attacks are related with integrity rather than confidentiality in Section 5.

3 Design

In this section, we explain our design of HTTPI sessions and cookies. HTTPI sessions are decided by the two parameters such as a session id and a session key. Every request and response includes a session id and the session id is associated with the session key during an HTTPI session establishment. The session id and the session key are managed by a session table in a web client and a web server. We define when to terminate the HTTPI session and introduce a cookie verifier to prevent cookies from being forged and replayed.

3.1 HTTPI Session

When a web client communicates with a web server using HTTPI, the web client establishes a session with the web server by the two parameters such as a session id, and a session key chosen by the web server. After defining the HTTPI session, the web client and the web server can send an HTTP request and an HTTP reply using HTTPI protocol in Figure 1. We devise two header fields such as the HMAC header field and the HMAC-control header field in Figure 1. They will be explained in detail in Section 6. The HMAC header field contains the session id value, the hash algorithm, and the hash value. The HMAC-control header field enables the HMAC header field by defining which header fields are included or excluded for hashing. Our first design was to compute the hash value for the entire page. Fortunately, $MD5(Content)$ is defined as Content-MD5[36], and the hash value of the HMAC header field can be computed with header fields and the session key by enabling the optional Content-MD5 header field.

3.2 HTTPI Session Establishment

We define that a web client and a web server establish an HTTPI session when they share a session key with each other. With the session key between the web client and the web server, the web client keyed-hashes an HTTP request with the session key and sends the HMAC header field with the HTTP request to the web server. The web server verifies the HMAC header field and can accept or reject the HTTP request. If the HTTP request is verified, the web server keyed-hashes an HTTP response with the session key, and sends the HMAC header field with the HTTP response to the web client. The web client verifies the HMAC header field, and can accept or reject the HTTP response.

In order to share a session key between the web client and the web server, we use TLS[6] to establish an HTTPI session between the web client and the web server. Though we adopt TLS to establish an HTTPI session in our approach, we can use Diffie-Hellman[7] or SRP[46] to exchange session keys.

3.3 Session Table

As HTTPS manages a session table in a web server, HTTPI requires a web server to manage a session table. The session table in the web server is illustrated in Figure 2. The amount of information in the session table in the web server is not as large as that in HTTPS. Similarly, the session table of a web client is illustrated in Figure 3. The primary key for the session table in a web client is Server URL, and a web client manages one HTTPI session per a web server. On the other hand, the primary key for the session table in a web server is a session id since the client IP address can not be used to identify the web client due to the dynamic IP address changes.

3.4 Session Termination

A web server can terminate a session in its session table depending on the expiration time and the number of sessions. First, if a session passes the expiration time in the session table, the session should be terminated. Second, if the number of sessions exceeds the maximum threshold, a web server should remove the session from the oldest ones. On the other hand, a web client does not maintain the expiration time in the session table. Thus, the web client has no explicit way to terminate an HTTPI session. However, if a web server receives an HTTP request from a web client and can not find the session id in the HTTP request in the session table, the web server should reply to the web client that the session is terminated, and the web client should terminate the session record relevant to the web server in the session table. If the web client wants to visit the website, the web client should re-establish an HTTPI session using HTTPS.

3.5 Session Resumption

HTTPI session resumption is different from HTTPS session resumption. Since an HTTPI session id is always sent in clear text using an HTTP header field, HTTPI session can be resumed without additional handshakes even though a web client is disconnected from a web server. On the other hand, HTTPS session can not be resumed without fast handshakes since the session id of

$B \rightarrow W$: $Host, Method, Request\text{-}uri, Cookie_{user}\{domain, path, secure\}, HMAC, HMAC\text{-}control$
$W \rightarrow B$: $status, MD5(Content), HMAC, HMAC\text{-}control$

Figure 1: HTTPI Protocol

Session Id	Session Key	Client IP Address	Port Number	Expiration Time

Figure 2: Session Table in a Web Server

Server URL	Session Id	Session Key	Server IP address

Figure 3: Session Table in a Web Client

TLS can be only seen when a web client and a web server exchange hello messages. After handshakes, every message over IP layer is encrypted in HTTPS.

3.6 Verified Cookie

Every cookie that is exchanged by HTTPI has to have at least three fields as follows: name, value, verifier, and other fields. We devise a cookie verifier to keyed-hash the value of a cookie as follows:

Verifier := H(K, SK, Value)

In our approach, we use server key K and session key SK to verify cookies. We use a server key to create unforgeable cookies by a web client and a session key to prevent cookies from being replayed by an attacker. If cookies have a verifier in the value of the cookies, we say that the cookies are verified. If the session in the session table is terminated, all the verified cookies relevant to the the session id can no longer be used such that the web client should also expire all the verified cookies.

4 Security Guarantees

In this section, we show security guarantees by HTTPI. We derive HTTPI security guarantees to resolve problems from P1 to P4 in Section 1. HTTPI provides server authentication, message integrity, and cookie integrity as security guarantees to address these problems. We use these security guarantees to explain how HTTPI defends against diverse web attacks in Table 2.

SG1. Server Authentication

From Problem P1 and P4, every request should be preceded by server authentication and we derive server authentication as follows:

> If web client B succeeds in establishing an HTTPI session supposedly with some web server W, then the established session is indeed between B and W.

When W sends an HTTP response to an HTTP request, W should use the session key corresponding to the session id in the HTTP request to compute the HMAC header field in the HTTP response. When B receives the HTTP response from W, B can authenticate W by verifying the HMAC header field with the session key associated with the session id in the HTTP response header.

SG2. Message Integrity from W to B

From Problem P2 and P4, integrity is required to protect users from web attacks and we derive message integrity from W to B as follows:

> If web client B receives a message supposedly sent by some web server W in some HTTPI session, then B can check that this message was indeed sent by W in this session.

If B receives a modified message in-transit from W to B, B can detect that the message is not originally sent from W since the HMAC of the message received from W is different from the HMAC of the message computed by B.

SG3. Message Integrity from B to W

Similarly, from Problem P2, we derive message integrity from B to W as follows:

> If web server W receives a request message that is supposedly sent (by some unknown web client) in some HTTPI session, then W can check that this message was indeed sent in the HTTPI session.

If W receives a modified request message in-transit from B to W, W can detect that the message is not originally sent from B since the HMAC of the message received from B is different from the HMAC of the message computed by W.

SG4. Cookie Integrity from W to B

From Problem P3, cookie should not be overwritten, and we derive cookie integrity from W to B as follows:

> If there is an established HTTPI session between web client B and web server W, and B receives some cookie c from W during this session, then any attempt by another web server W' to send another cookie c' to overwrite the stored c in B will fail.

If W' tries to overwrite c with c', B can verify c' by using the session key associated with the session id in the response, and computing the HMAC of cookie contents with the session key. We discuss about this type of cookie as the cookie verifier in detail in Section 3.

SG5. Cookie Integrity from B to W

From Problem P3, cookies should not be replayed and we derive cookie integrity from B to W as follows:

> If web server W receives some verified cookie c from web client B while there is an established HTTPI session between B and W, then W can check whether c has been sent earlier from W to B during the same session.

If an attacker eavesdrops or steals c from B to W, and the attacker replay c to W, W can verify c using the session key associated with the session id in the request and computing the HMAC of cookie contents with the session key.

5 Attacks

In this section, we analyze various web attacks and categorize them as *attacks defended by HTTPI* and *attacks not defended by HTTPI*. Surprisingly, HTTPI can defend against many existing attacks, and a new attack like DNS cache poisoning[44] and SSLStrip[32]. Thus, HTTPI is coherent in defending against existing attacks and future attacks, which can be categorized in our analysis as in Table 2.

5.1 Attacks Defended by HTTPI

For attacks defended by HTTPI, we classify them as server impersonation, Man-in-the-Middle (MITM), message modification, cookie injection, and cookie theft in Table 2. We show how HTTPI protects users from these categories of web attacks using security guarantees in Section 4.

Server Impersonation

Server impersonation attacks make use of the lack of any binding whether it is a binding between Domain Names and IP addresses, or it is a binding between MAC Address and IP address. An attacker provides the IP address of an illegitimate website instead of that of a legitimate website by compromising Home Routers, rebinding DNS entries with a Delegated DNS or poisoning a DNS cache. For example, Drive-By Pharming[43, 39, 38], DNS Rebinding[18], and DNS cache poisoning[44] are server impersonation attacks.

Message Modification

Message modification attacks make use of the lack of message integrity in HTTP requests and HTTP responses. For example, HTTP messages can be easily modified by many entities in the Internet including ISP providers, proxies, and gateways. ISP providers change pages in-flight with injected advertisements to increase revenues. On the other hand, proxy servers in some organizations filter advertisements and pop-ups to get rid of annoyances by users[40]. ARP poisoning can be used as a middleman to modify messages between browsers and web servers.

Cookie Injection

The risks of cookie injection have been known from 2004[19], and cookie injection can be categorized as *cross-domain cookie injection* and *cross security boundary cookie injection* as follows:

Cross-domain cookie injection: Cross-domain cookie injection takes advantage of the length of top level domains in the case of country domains, and this vulnerability is fixed by defining the minimum length for each domain. As we saw cookie SOP in Section 1, cookie SOP requires *longest suffix rule* for the domain attribute. Precisely, *longest suffix rule* for the domain attribute requires *minimum suffix rule* so that the domain attribute of a cookie should not be ambiguous. For example, if the top level domain is not a country domain such as `.com`, the domain attribute of a cookie should include at least the 2nd level domain specific name such as `example.com`. If the top level domain is a country domain such as `.kr`, the domain attribute of a cookie should include at least the 3rd level domain specific name such as `example.co.kr`. However, cross-domain cookie injection is still effective in the current web architecture if an attacker can impersonate any website with a longer or equal domain suffix as a

Attacks	Examples of Attacks	HTTPI Defenses against Attacks
Server Impersonation	Drive-By Pharming[43, 39, 38], DNS Rebinding[18], DNS cache poisoning[44]	SG1. Server Authentication
Man-in-the-Middle	SSLStrip[32]	SG1. Server Authentication, SG2. Message Integrity from W to B
Message Modification	In-flight Page Change[40], ARP poisoning[49, 47]	SG2. Message Integrity from W to B, SG3. Message Integrity from B to W
Cookie Injection	Session Fixation[29]	SG4. Cookie Integrity from W to B
Cookie Theft	Side Jacking[15], Surf Jacking[14]	SG5. Cookie Integrity from B to W

Table 2: Classification of Web Attacks and Defenses by HTTPI

target domain.

Cross-security boundary cookie injection: Cross security boundary cookie injection leverages the lack of cross security boundary policy in terms of cookies. The secure attribute of cookies restricts information to flow from a higher security level, HTTPS to a lower security level, HTTP. For example, if cookies are set to be *secure*, Secure cookies can not be read by HTTP, but can be read only by HTTPS. This policy effectively prevents Secure cookies from being leaked over HTTP. However, there is no protection for cookies set over a lower-level security, HTTP, to be used over a higher-level security, HTTPS.

Cookie Theft

Cookies provide the state of browsers from trivial user information to critical authentication information. If an attacker steals a user's authentication cookie, the attacker can log in a website as the user. For example, Side Jacking[15] and Surf Jacking[14] are two examples of cookie theft. Side Jacking is a passive cookie theft relying on eavesdropping while Surf Jacking is an active cookie theft by redirecting a user's browser to any target website from which an attacker wants to harvest cookies with message modification. These two attacks can be effectively mitigated by setting cookies with the secure attribute. However, it is not always implemented in practice due to *cross-domain redirection* and *cross-security redirection*. Many websites use multiple web servers to serve their users due to scalability. If `login.example.com` sets cookies with the domain attribute, `domain=login.example.com`, the cookies can not be used for `www.example.com`. Thus, cookie leaking can be minimized by giving a more specific domain name to the domain attribute in cookies. However, if `example.com` wants to serve users with multiple web servers, `login.example.com` should set cookies with `domain=example.com`. If a cookie is set with the domain attribute, `domain=example.com`, an attacker just needs to register a domain name with `evil.example.com` in a delegated DNS server. If the attacker can attract a user to visit `evil.example.com`, the attacker can harvest cookies belong to `example.com`. Many websites serve users with HTTP after users log in websites with HTTPS due to the performance degradation by TLS. While users log in a website with HTTP, authentication cookies can be stolen by attackers if authentication cookies are not set to be *secure*. However, if a website would like to use an authentication cookie to redirect a user from HTTPS to HTTP, the authentication cookie must be read by HTTP and that makes an authentication cookie not *secure*.

5.2 Attacks Not Defended by HTTPI

For attacks not defended by HTTPI, we explain why HTTPI can not defend against these attacks. Interestingly, attacks not defended by HTTPI are not defended by HTTPS, either.

Phishing: Phishing attacks[8] lures users by emails to visit illegitimate websites having the same look and feel and steal credentials from users. Phishing misleads human perceptions to illegitimate websites by masquerading URLs or binding domain names to invalid certificates. HTTPI can not defend against Phishing attacks like HTTPS since Phishing attacks leverage the lack of binding between legitimate websites and human perceptions. If a user perceives an illegitimate website as a legitimate website with a masqueraded URL or an invalid certificate, HTTPI is not effective since HTTPI can only protect the user from illegitimate websites impersonating legitimate websites with original URLs. Similarly, HTTPS can not defend against Phishing attacks if a certificate is issued to illegitimate websites

or if a user accepts invalid certificates.

Cross Site Request Forgery (CSRF): CSRF attacks[48] make use of the trust that a website has in a user. If a user authenticates to a website and establishes a trust between the user and the website, an attacker can make use of the trust that the website has in the user's browser and delegates unauthorized commands of the attacker to the website with the user's privilege. The website can check whether HTTP requests come from the browser, which established the HTTPI session between the web server and the browser, but can not check whether HTTP requests are authorized by the user or not. Similarly, HTTPS can not protect the user from CSRF attacks since the web server can not check whether HTTP requests are authorized by the user or not.

Cross Site Scripting (XSS): XSS attacks[26] make use of the trust that a user has in a website. If a web site authenticates a user and establishes a trust between the website and the user, an attacker can make use of the trust that the user has in the website and delegates unauthorized commands of the attacker to the user with the website's privilege. The browser can check whether HTTP responses come from the website, which established the HTTPI session between the browser and the web server, but can not check whether HTTP responses are authorized by the web server. Similarly, HTTPS can not protect the user from XSS attacks since the web server can not check whether HTTP responses are authorized by the web server or not.

6 Implementation

We explain implementation details of HTTPI in this section. Hashing header fields and contents seems trivial, but the devils are in detail.

6.1 Content Hashing

The Content-MD5 header field is defined as the MD5 hash of an entity-body[36] as follows:

Content-MD5 := H(entity-body)

An entity-body is any content-type data applied with some encoding such as compression[9] as follows:

entity-body := Content-Encoding(Content-Type(data))

If transfer-coding is applied, it becomes a message-body used to carry the entity-body associated with an HTTP request or response[9]. The Content-MD5 header field should be applied to a content after some content encoding, but before some transfer encoding. This definition does not address instance manipulations like range-selection or delta encoding and the concept of instance[33] is introduced. Precisely, the Content-MD5 header field should be applied to a content after some content encoding and before some instance manipulations or some transfer encoding. More precisely, if we consider dynamic contents by server-side scripts, the Content-MD5 header field should be applied to a content after some content encoding and the execution of server-side scripts, and before some instance manipulations or some transfer encoding. Currently, Apache 2.2.11 computes Content-MD5 for static contents and we implemented the filter to compute the Content-MD5 header field for dynamic contents.

6.2 Decoupling Header and Contents

Our first design of HTTPI keyed-hash entity-header fields and an entity-body together. However, it becomes clear shortly that header fields and a content should be decoupled for hashing due to the following two reasons. First, it is inflexible since it still can not support caching even without encryption, and it has no difference from using TLS without cipher only. In fact, TLS supports a null cipher feature such as TLS_RSA_WITH_NULL_SHA or TLS_RSA_WITH_NULL_MD5[6] though they are not used in practice. Second, it is inefficient since it hurts the pipelining of a web server. The web server can generate the HMAC header field of an instance after reading the instance completely.

Our second design of HTTPI separates header fields from an instance and we use the Content-MD5 header field[36] for contents hashing and keyed-hash header fields with the Content-MD5 header field. It is advantageous in many ways compared to our initial design of HTTPI. First, it is flexible to support caching since the value of contents hashing does not change unless the contents change. Contents hashing can be cached or precomputed if a webpage is static. If the Content-MD5 header field is computed for a static content initially, the Content-MD5 header field can be used for other users. Dynamic webpages can be made possible with two technologies such as client-side scripts and server-side scripts. Since client-side scripts are executed in a browser, the contents hashing needs to consider only server-side scripts for a dynamic webpage. A webpage consists of many web objects including images, stylesheets, and scripts. Contents hashing can be precomputed and cached for images and stylesheets always. Client-side scripts can also be precomputed and cached. Header fields contain more specific information

including date, cookies and sometimes authentication. Thus, contents are most likely user-independent and header fields are user-dependent, and it is reasonable to separate header fields and contents from an instance for hashing. In addition to that, if the Content-MD5 header field is replaced by the Content-SHA1 header field, which does not exist currently, in the future due to the weakness of MD5, the logic of HTTPI needs not to be changed. Second, it is efficient not to hurt the pipelining of a web server since it only requires to compute the HMAC of header fields and add the HMAC header field as the last header field on the fly instead of waiting for the computation of keyed-hashing of contents as in our first design.

6.3 Our New Header Fields

We design two header fields for HTTPI: 1) HMAC 2) HMAC-control. We illustrate the HMAC and the HMAC-control header field in Figure 4 and Figure 5, respectively. We follow the definitions of Augmented BNF in [9]. *Method*, *Request-URI*, and *Status-Code* in Figure 5 follows the definitions in [9]. The HMAC header field contains the session-id value, the hashing algorithm such as md5 or sha1 and the hash value of header fields with the session key, SK.

An HTTP request consists of the request-line, the request header, and the body. Similarly, An HTTP response consists of the status-line, the response header, and the body. The request-line and the status-line can be changed by proxies and should not be used for keyed-hashing directly. Moreover, the request-line and the status-line processing must be tolerant in a web server and a browser since they can contain extra spaces and tabs[9]. However, the values used by an origin server should be kept since these values can be modified for attacks. The request-line consists of the method, the request-uri, and the http-version, and the status-line consists of the http-version, the status-code, and the reason-phrase. Thus, we add the method, the request-uri, the http-version, and the status-code in the HMAC-control header field. Additionally, we create the *must-include* and the *must-exclude* header fields for the HMAC-control header field. If a 304 (Not Modified) response is used by an origin web server,the cache may include more header fields other than the header fields received by the origin web server. In this case, the origin web server can enumerate all the header fields to compute the HMAC in the *must-include* header field when an HTTP response is received by a browser. If an origin web server is clockless, the origin web server does not generate the Date header field, and proxies may add the Date header field to the header. In this case, the origin web server can use the *must-exclude* header field to note that the origin web server does not generate the Date header field.

6.4 Caching in HTTPI

In order to support the caching mechanism in HTTP, HTTPI is required to keyed-hash header fields selectively. There are two kinds of header fields depending on the behavior of caching: end-to-end header fields and hop-by-hop header fields. The following HTTP/1.1 header fields are hop-by-hop headers: Connection, Keep-Alive, Proxy-Authenticate, Proxy-Authorization, TE, Trailers, Transfer-Encoding and Upgrade[9]. All the other header fields defined by HTTP/1.1 are end-to-end header fields. End-to-end header fields should be included for computing the HMAC header field, but hop-by-hop header fields should be excluded. Other hop-by-hop header fields must be listed in the Connection header field to be introduced into HTTP/1.1 or later[9] and these header fields should be excluded, too. We found that Via and Warning header fields can be modified in-transit and they should be excluded for computing the HMAC header field.

Some proxies might convert original contents to some other new formats and can break the Content-MD5 header field. There are two types of proxies such as a transparent proxy and a non-transparent proxy. A transparent proxy passes requests and responses unmodified whereas a non-transparent proxy modifies requests and responses to convert between image formats for saving cache space or reducing the amount of traffic. Unfortunately, if a non-transparent proxy convert original contents to some other new formats, HTTPI can not work since the Content-MD5 header field will be different for a new format. If an HTTP message includes the *no-transform* directive, the cache or the proxy should not change any aspect of the entity-body specified by the Content-Encoding, the Content-Range, and the Content-Type header field including the entity-body itself[9]. Thus, *no-transform* should be used with HTTPI.

When a cache makes a request to an origin web server, and the origin web server provides a 304 (Not Modified) response or a 206 (Partial Content) response, the cache then constructs a response and send the response to a browser. The 304 response from the origin web server contains only header fields and the cache retrieves the entity-body stored in the cache entry and combine the header fields and the entity-body to construct an HTTP response to the browser. The origin web server can still use HTTPI for this caching protocol if the origin web server includes the Content-MD5, the HMAC and the HMAC-control header field. Since the HMAC header field is computed only with header fields, the origin web server can compute the HMAC header field and the ori-

HMAC: session-id=quoted-string, alg= md5 | sha1, hash=H(SK,1#header-field)

Figure 4: HMAC Header Field

HMAC-control: http-version=1*DIGIT, method= Method, request-uri=Request-URI, status=Status-Code, must-include=*(header-field), must-exclude=*(header-field), nonce=quoted-string

Figure 5: HMAC-control Header Field

gin web server needs to enumerate all the header fields to compute the HMAC header field in the *must-include* header field since the cache may include more header fields than the header fields provided by the origin web server. When the cache receives the 304 response from the origin web server, the cache can combine the entity-body in the cache entry as usual. When the browser receives the HTTP response from the cache, the browser can check the HMAC header field by computing all the enumerated header fields in the *must-include* header field and the Content-MD5 of the entity-body. The request to the 304 response includes the If-Modified-Since or the If-None-Match header field to check whether objects are modified or not after the browser receives the content previously. The If-Modified-Since header field is based on the Last-Modified header field and check whether the objects are modified from the date in the Last-Modified header field. The If-None-Match header field depends on the ETag header field and the origin web server should ensure that the ETag header field is uniquely changed whenever a content is changed. In both cases, the origin web server might not be able to generate the Last-Modified or the ETag header field if web objects are generated dynamically from a database. It is difficult to know when the objects are generated and how the objects have a unique ETag if they are generated dynamically. Due to these limitations, Nottingham proposes to use the Content-MD5 header filed for a strong cache validation with a new header field called *If-Not-Hash* instead of the If-Modified-Since and the If-None-Match[37] header field. Moreover, MD5 hash can be used to detect duplicate transfer[34]. A traditional web cache indexes each entry by a given URL, but this can cause a redundant payload transfer by a cache miss between proxies and origin web servers. Therefore, Content-MD5 can be beneficial not only for integrity but also for performance.

7 Performance

7.1 Evaluation

We have implemented HTTPI as a module in Apache version 2.2.11 and tested the performance by modifying httperf version 0.9.0[35]. The first session of HTTPI is established using HTTPS, and the rest of the sessions use

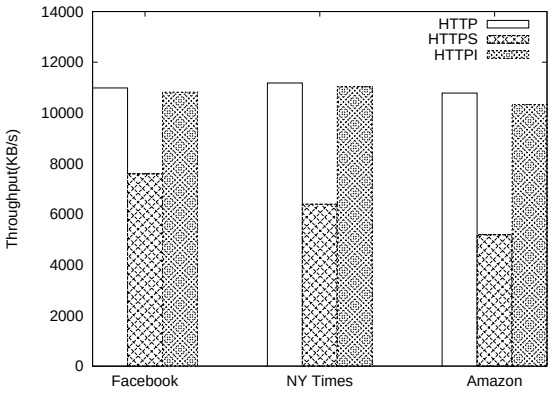

Figure 6: Throughput of HTTP, HTTPS, and HTTPI in 100 Mbps Ethernet

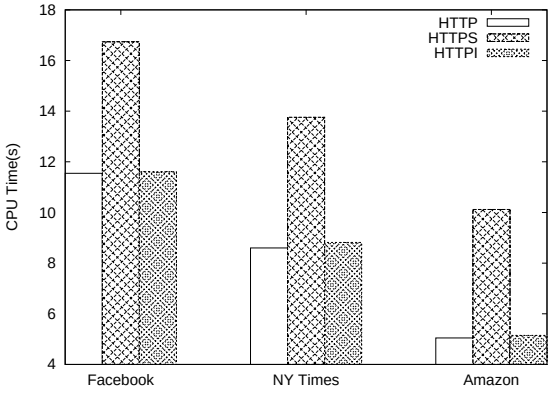

Figure 7: CPU Time of HTTP, HTTPS, and HTTPI in 100 Mbps Ethernet

HTTP with keyed-hashing HTTP header fields including the Content-MD5 header field. In the first request for each content, the hash of the content is computed and cached. In the subsequent requests, the hash of contents is not computed if it is found in the cache, but is retrieved from the cache. We have chosen Facebook, the New York Times, and Amazon for our experiments. Social networking websites like Facebook serve users with HTTPS for authentication, and users are redirected to HTTP. News website like the New York Times do not deploy HTTPS and serve users only with HTTP. Online shopping websites like Amazon serve users with HTTP

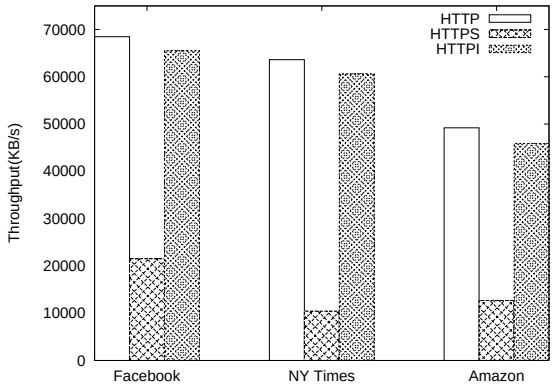

Figure 8: Throughput of HTTP, HTTPS, and HTTPI in 1 Gbps Ethernet

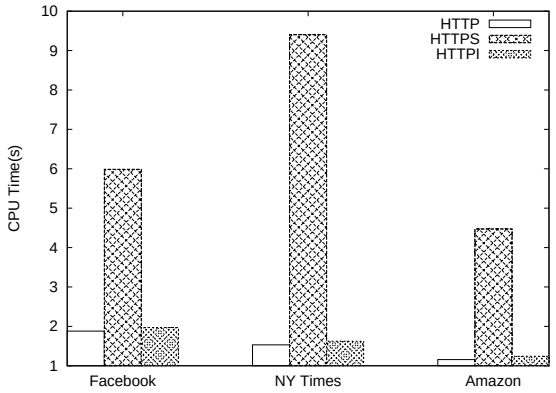

Figure 9: CPU Time of HTTP, HTTPS, and HTTPI in 1 Gbps Ethernet

after authentication and before checkout while users are browsing the website. One of our authors have scraped his personal page from Facebook, the index page from the New York Times, and his main page from Amazon. The Facebook page consists of a container HTML page about 360 KB and 54 files of images, scripts, and stylesheets about 1.1 MB. The New York Times page consists of a container HTML page about 140 KB and 94 files of images, scripts, and stylesheets about 1.1 MB. Amazon page consists of a container HTML page about 172 KB and 54 files of images, scripts, and stylesheets about 484 KB.

We have used two Linux machines with Ubuntu 8.04 using Intel Pentium 4 3.40 GHz for a web client and Intel Core 2 2.13 GHz for a web server both with 2GB of RAM. Each machine is connected with 100 Mbps link in LANs and 1 Gbps fiber backbone between them. We have measured server throughput, and CPU time using httperf by sending GET requests to the web server for 100 sessions. We ran 10 trials for the experiment and show our results on average in Figure 6, and Figure 7. The server throughput is capped by the bandwidth of 100 Mbps. Though this environment might be more realistic, we have used Gigabit Ethernet for further measurements to see the performance difference between HTTP and HTTPI. We have used two Linux machines with Ubuntu 8.04 using Intel Core 2 Duo 3 GHz for a web client and a web server both with 2 GB of RAM connected with Gigabit Ethernet. Figure 8 and Figure 9 show our results. In both environments, HTTPI outperforms HTTPS and shows less than 10 % performance degradation in all cases compared to HTTP.

7.2 Discussion

The performance of HTTPI should be considered in the context of caching as well. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases for latency reduction and to eliminate the need to send full responses for bandwidth reduction, and HTTPI supports it like HTTP with integrity. In terms of bandwidth, caching proxies and contents servers can be still supported and used to reduce bandwidth with HTTPI. Since the HTTP response does not have to come all the way from the origin web server, the bandwidth will be reduced at a web server. In terms of latency, propagation delays can be reduced by placing caching servers close to users. Queuing delays can also be reduced by precomputing or caching contents. The Contents-MD5 header field can be precomputed and cached for static contents. Many studies have shown that dynamic contents can also be cached[16, 50, 4].

8 Related Work

8.1 Message Integrity Techniques

HTTP provides the Content-Length[9] header field and the Content-MD5[36] header field as message integrity techniques. The Content-Length header field indicates the size of the entity-body and it can be used to detect if HTTP pages are modified. However, it is not difficult to modify HTTP pages given the Content-Length header field and the Content-Length header field itself can be forged or omitted. The Content-MD5[36] header field can be more reliable than the Content-Length header field since it is hard to find a collision for the Content-MD5 header field provided that the Content-MD5 header field can not be modified or omitted. The Content-MD5 header field is an optional header field and not widely used due to the overhead of MD5 computation. We found that using the Content-MD5 header field is helpful for HTTP page integrity and HTTP caching coherence. In our approach, we have decoupled a content hashing and

header hashing by readily adopting the Content-MD5 header field. With the Content-MD5 header field, HTTPI uses HMAC[30] to keyed-hash HTTP header fields and HTTP pages can not be modified without being detected. Web Tripwire[40] is an integrity mechanism to detect the modification of HTTP pages in web applications by comparing requested HTTP pages with the known good representations of requested HTTP pages using a tripwire script. The detection mechanism of Web Tripwire is limited to received contents, especially only an HTML page and there is no page integrity from a client to a web server. In addition to that, there is no message integrity about HTTP redirection or error messages though these redirection or error message are easily used by attackers to trick users. Web Tripwire requires more bandwidth for the known good representations about 17 %. HTTPI requires two more header fields and the size of these header fields does not vary depending on the size of contents, but Web Tripwire requires more bandwidth depending on the size of contents. Nevertheless, Web Tripwire is not cryptographically secure and a false positive and a false negative can occur. On the other hand, HTTPI provides a complete solution for message integrity in HTTP pages without a false positive and a false negative. HTTPI provides message integrity for both HTTP requests and HTTP responses and for any web objects. Saltzman and Sharabani proposes HTTP Response Signing[42], but signing requires more computation than hashing and it only protects HTTP responses.

8.2 IPsec Analogy

HTTPI is analogous to IPsec[24] in terms of design. HTTPI provides integrity as IPsec provides integrity with IP Authentication Header (AH)[22]. On the other hand, HTTPS provides both integrity and confidentiality as IPsec provides both with IP Encapsulation Security Payload (ESP)[23].

While IPsec[24] is a protection mechanism for the IP layer, HTTPI is a protection mechanism for the Application layer and especially HTTP. HTTPI uses a session id to identify a corresponding session key for each session. Using the session id in HTTP header fields is similar to the Security Parameter Index (SPI) in IPsec. HTTPI uses a session cache like the security database in IPsec. Since HMAC is a keyed hashing Message Authentication Code (MAC), browsers and web servers need to share keys. Keys can be shared by using out-of-band methods or appropriate cryptographic protocols. HTTPI uses HTTPS to share keys between web clients and web servers. IPsec provides two ways to share keys such as manual configuration and Internet Key Exchange (IKE)[21]. Key sharing with IKE in IPsec is similar to that with HTTPS in HTTPI.

8.3 Partial Solutions

There are several proposals to solve the problems from P1 to P4 in Section 1 but none of them address the security guarantees of Section 4 like HTTPI totally.

Server Impersonation: server impersonation attacks are possible by many ways such as Pharming[43, 39, 38], DNS rebinding[18], DNS cache poisoning[44], and ARP poisoning[49, 47]. Locked Same Origin Policy (LSOP)[20] is proposed to solve Dynamic Pharming attacks. LSOP assumes that users accept invalid certificates in Dynamic Pharming attacks. If accepting invalid certificate are allowed as an assumption, it could cause more problems. Furthermore, LSOP solutions can not address Dynamic Pharming attacks when it comes to use HTTP. Any solutions to defend against DNS misbinding such as Pharming, DNS rebinding, and DNS cache poisoning can not be used to defend against MAC address misbinding by ARP poisoning and vice versa. Furthermore, any one solution of the set of problems in server impersonation attacks can not address another problem in the same set of problems. HTTPI can address the set of problems like server impersonation attacks with server authentication.

Header Modification: HTTP headers are consistently attacked whereas HTTP header fields are proposed to defend against web attacks. The HTTP Referer header field can be used to mitigate CSRF attacks, but the HTTP Origin header field[2] is proposed due to the privacy leaks by the HTTP Referer header field. More header fields can be proposed to ensure security in the future, but these header fields can be modified without any proper header protection. HTTPI provides message integrity for HTTP header fields, and HTTPI can be complementary to many proposals for security relying on HTTP header fields.

Cookie Theft: several approaches are proposed for cookie protection. SessionLock[1] protects users from cookie eavesdropping. SessionLock secures web sessions from SideJacking[15] using a session secret shared between a browser and a web server over TLS. The browser uses the session secret to authenticate to the web server using the HMAC of timestamp and the request URL in every subsequent HTTP request. Fu et al.[13] proposes to use a server key to protect cookies from being forged, but the cookies using a server key can be replayed until the cookies expire. HTTPI defends against cookie eavesdropping by the cookie verifier and protects cookies from being forged and replayed. If a server key is only used, it protects cookies from being forged, but does not prevent cookies from being

replayed. HTTPI protects cookies from being replayed by using a session key.

HTTP-To-HTTPS redirection: the HTTP-to-HTTPS redirection problem is addressed by ForceHTTPS[17]. If ForceHTTPS cookie is set or configured by a user, the user is redirected to HTTPS by URL rewrite rules. ForceHTTPS is a complementary mechanism for the error processing mechanism of browsers for HTTPS. On the other hand, HTTPI is a complementary approach to use HTTP. ForceHTTPS can be used to solve SSLStrip[32] like HTTPI.

9 Conclusion

By observing and analyzing the types of attacks such as server impersonation, message modification, cookie theft and cookie injection, we have found that these categories of attacks are related to integrity rather than confidentiality. To counter these types of threats, we have designed a lite protocol for secure web, HTTPI. By doing so, we have decoupled HTTP headers and contents for hashing and support the caching mechanism of HTTP. Thus, HTTPI is scalable like HTTP. We have shown that the Content-MD5 header field is useful for caching and security. The Content-MD5 header field is an optional header field and not widely used by many websites. We claim that any website reluctant to adopt the Content-MD5 header field should rethink that the computational power to compute the Content-MD5 header field is not wasted. Rather, it can be payed with a bigger reward in terms of caching and security.

We have used realistic replicas of websites such as Facebook for social networking, the New York Times for news, and Amazon for online shopping for our experiment and have shown that HTTPI outperforms HTTPS in terms of throughput, and CPU time. We have shown less than 10 % performance degradation compared to HTTP. HTTPS is overkill for these types of websites and we envision that they will willingly deploy HTTPI for security without performance penalty.

In our extensive research, we claim that we should adopt HTTPI in light of performance, usability, and security. In performance, HTTPI does not degrade the performance of HTTP and outperforms HTTPS in terms of bandwidth and latency. In usability, HTTPI is transparent to users compared to HTTP Authentication. In security, HTTPI provides reasonable security. Most importantly, we should adopt HTTPI as a secure framework for the future web architecture. since HTTPI does not break the current web architecture and HTTPI supports the distributed information system of web architecture by allowing caching. Thus, HTTPI provides performance and scalability with security – its security is comparable to HTTPS, while its performance and scalability is comparable to HTTP.

References

[1] ADIDA, B. Sessionlock: securing web sessions against eavesdropping. In *WWW '08: Proceeding of the 17th international conference on World Wide Web* (New York, NY, USA, 2008), ACM, pp. 517–524.

[2] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *CCS '08:Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)* (2008).

[3] BERNERS-LEE, T., FIELDING, R., AND FRYSTYK, H. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[4] BOUCHENAK, S., COX, A., DROPSHO, S., MITTAL, S., AND ZWAENEPOEL, W. Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution. In *ACM/IFIP/USENIX 7th International Middleware Conference (Middleware-2006)* (Melbourne, Australia, Nov. 2006).

[5] CHEN, S., MAO, Z., WANG, Y.-M., AND ZHANG, M. Pretty-bad-proxy: An overlooked adversary in browsers https deployments. In *In Proceedings of the 2009 IEEE Symposium on Security and Privacy* (2009).

[6] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008.

[7] DIFFIE, W., AND HELLMAN, M. E. New directions in cryptography. *IEEE Transactions on Information Theory IT-22*, 6 (1976), 644–654.

[8] FELTEN, E. W., BALFANZ, D., DEAN, D., AND WALLACH, D. S. Web spoofing: An internet con game. In *20th National Information Systems Security Conference (Baltimore, Maryland)* (October 1997).

[9] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[10] FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., AND STEWART, L. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999.

[11] FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LEACH, P., LUOTONEN, A., SINK, E., AND STEWART, L. An Extension to HTTP : Digest Access Authentication. RFC 2069 (Proposed Standard), Jan. 1997. Obsoleted by RFC 2617.

[12] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with coral. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 18–18.

[13] FU, K., SIT, E., SMITH, K., AND FEAMSTER, N. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium* (Washington, D.C., August 2001). An extended version is available as MIT-LCS-TR-818.

[14] GAUCI, S. Surf jacking - "https will not save you". `http://enablesecurity.com/2008/08/11/surf-jack-https-will-not-save-you`, August 2008.

[15] GRAHAM, R. Sidejacking with hamster. http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html, August 2007.

[16] IYENGAR, A., AND CHALLENGER, J. Improving web server performance by caching dynamic data. In *In Proceedings of the USENIX Symposium on Internet Technologies and Systems* (1997), pp. 49–60.

[17] JACKSON, C., AND BARTH, A. Forcehttps: protecting high-security web sites from network attacks. In *WWW '08: Proceeding of the 17th international conference on World Wide Web* (New York, NY, USA, 2008), ACM, pp. 525–534.

[18] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting Browsers from DNS Rebinding Attacks. In *In Proceedings of ACM CCS 07* (2007).

[19] JOHNSTON, P., AND MOORE, R. Multiple browser cookie injection vulnerabilities. http://www.westpoint.ltd.uk/advisories/wp-04-0001.txt, September 2004.

[20] KARLOF, C., SHANKAR, U., TYGAR, J. D., AND WAGNER, D. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 58–71.

[21] KAUFMAN, C. Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard), Dec. 2005. Updated by RFC 5282.

[22] KENT, S. IP Authentication Header. RFC 4302 (Proposed Standard), Dec. 2005.

[23] KENT, S. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), Dec. 2005.

[24] KENT, S., AND SEO, K. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), Dec. 2005.

[25] KHARE, R., AND LAWRENCE, S. Upgrading to TLS Within HTTP/1.1. RFC 2817 (Proposed Standard), May 2000.

[26] KLEIN, A. Cross site scripting explained. Sanctum Security Group, June 2002.

[27] KLEIN, A. Exploiting the xmlhttprequest object in ie–referrer spoofing and a lot more... http://www.cgisecurity.com/lib/XmlHTTPRequest.shtml, September 2005.

[28] KLEIN, A. Forging http request headers with flash actionscript. http://www.securiteam.com/securityreviews/5KP0M1FJ5E.html, July 2006.

[29] KOLŠEK, M. Session fixation vulnerability in web-based applications. www.acros.si/papers/session_fixation.pdf, December 2002.

[30] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997.

[31] KRISTOL, D., AND MONTULLI, L. HTTP State Management Mechanism. RFC 2965 (Proposed Standard), Oct. 2000.

[32] MARLINSPIKE, M. New techniques for defeating ssl/tls. http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf, February 2009.

[33] MOGUL, J., AND HOFF, A. V. Instance Digests in HTTP. RFC 3230 (Proposed Standard), Jan. 2002.

[34] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in http. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 4–4.

[35] MOSBERGER, D., AND JIN, T. httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev. 26*, 3 (1998), 31–37.

[36] MYERS, J., AND ROSE, M. The Content-MD5 Header Field. RFC 1864 (Draft Standard), Oct. 1995.

[37] NOTTINGHAM, M. Inherent http coherence. http://www.mnot.net/papers/coherence.html.

[38] RAMZAN, Z. Dns pharming attacks using rogue dhcp. http://www.symantec.com/connect/blogs/dns-pharming-attacks-using-rogue-%dhcp, December 2008.

[39] RAMZAN, Z. Drive-by pharming in the wild. http://www.symantec.com/connect/blogs/drive-pharming-wild, January 2008.

[40] REIS, C., GRIBBLE, S. D., KOHNO, T., AND WEAVER, N. C. Detecting in-flight page changes with web tripwires. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 31–44.

[41] RESCORLA, E. HTTP Over TLS. RFC 2818 (Informational), May 2000.

[42] SALTZMAN, R., AND SHARABANI, A. Active man in the middle attacks. OWASP AU 2009, Feburuary 2009.

[43] STAMM, S., RAMZAN, Z., AND JAKOBSSON, M. Drive-by pharming. In *ICICS* (2007), pp. 495–506.

[44] US-CERT. Multiple dns implementations vulnerable to cache poisoning. http://www.kb.cert.org/vuls/id/800113.

[45] VENNERS, B. Http authentication woes. http://www.artima.com/weblogs/viewpost.jsp?thread=155252, April 2006.

[46] WU, T. The secure remote password protocol. In *In Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium* (1997), pp. 97–111.

[47] ZDRNJA, B. Massive arp spoofing attacks on web sites. http://isc.sans.org/diary.html?storyid=6001, March 2009.

[48] ZELLER, W., AND FELTEN, E. W. Cross-site request forgeries: Exploitation and prevention. Tech. rep., Department of Computer Science, Center for Information Technology Policy, Princeton University, October 2008.

[49] ZHANG, K. Arp spoofing http infection malware. http://securitylabs.websense.com/content/Blogs/2885.aspx, December 2007.

[50] ZHU, H., , AND YANG, T. Class-based cache management for dynamic web content. In *IEEE INFOCOM* (2001), pp. 1215–1224.