HTTP Digest Integrity

Another look, in light of recent attacks

Version 1.0

VSR

Timothy D. Morgan

January 5, 2010

Contents

Introduction 1

Overview of HTTP Digest Authentication 1

RFC 2069 Mode 1

auth Mode 2

auth-int Mode 2

Related Attacks 3

SSL/TLS Renegotiation 3

HTTP Request Smuggling 4

Mitigation with Digest Authentication 6

Limitations 6

Conclusion 7

Acknowledgements 7

References 8

Introduction

Recent history has proven that web communications security is highly lacking in redundancy. That is, simple breaks in common protocols, such as SSL/TLS or the authentication mechanisms which support it, often lead to catastrophic gaps in security. Recent examples of this fragile architecture abound [0Prfx,DebKeys,MD5CA,OCSP3,ReTLS], and even when protocols and implementations themselves are sound, research indicates browser user interfaces continue to leave room for serious attacks [SSLWarn].

This paper explores how the seldom-used HTTP digest authentication protocol can be used to mitigate certain recent forms of attack, including SSL/TLS renegotiation and some types of HTTP request smuggling.

Overview of HTTP Digest Authentication

HTTP digest authentication [RFC2617] is designed as an incremental improvement to HTTP basic authentication, where user credentials are simply sent in cleartext as a base64 encoded value. HTTP digest authentication is not intended as a generally secure cryptographic protocol. In fact, if one assumes an attacker can fully control the communications channel between a client and server, digest authentication is vulnerable to man-in-the-middle attacks, downgrade attacks, and likely other weaknesses. Digest authentication also does not provide secrecy of HTTP headers or payloads, which is a primary requirement for many secure applications.

What digest authentication does provide is relative secrecy of user credentials. Instead of sending cleartext passwords, cryptographic hashes (digests) are sent which are generated based on the user password along with several other values. In this way, passive and active attackers cannot easily take the credentials of a naïve user (who reuses passwords on multiple sites) and attack other systems. In fact, digest authentication also incorporates web site specific information into the hash, preventing trivial reuse of hashes on different websites. With some optional replay prevention mechanisms, reuse of hashes on the same website can even be limited or avoided.

Two perhaps lesser known features of HTTP digest authentication are its integrity protection and server authentication. The level of these protections varies depending upon the optional features supported by clients and servers. These optional features are used in three primary "quality of protection" modes: legacy or RFC 2069 mode, "`auth`" mode, and "`auth-int`" mode.

RFC 2069 Mode

This mode is compatible with [RFC2069] and is used when a server does not advertise a quality of protection (`qop=...`) flag in the initial challenge. In this mode, the following primary values are used to compute a one-way hash sent back to a server:

- Username of authenticating user
- Password of the authenticating user
- HTTP authentication realm
- Server-provided nonce value from the challenge response
- URI of this request
- Method of this request (GET, POST, etc)
- Digest of the request body (optional)

Note that some additional items may be included in these hashes, but are not deemed particularly relevant for this discussion.

Clearly, even in this early revision of the protocol, certain elements are integrity protected (HTTP method and URI) and the body of a request can be protected, optionally.

VSR COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

auth Mode

RFC 2617 introduced revisions to digest authentication which allow for improved efficiency and authentication of servers. When servers advertise the `auth` or `auth-int` quality of protection modes, clients should select one of these (as opposed to the RFC 2069 mode) and some additional information is sent along to the server (and protected by cryptographic hashes):

- Nonce count
- Client nonce

The nonce count value is intended to provide an efficient way to mitigate replay attacks against the same server without having to refresh the server nonce. The client nonce is designed to allow clients to authenticate server identity by ensuring that servers know the user's password digest (proven in a later response header).

auth-int Mode

The `auth-int` mode is very similar to the `auth` mode, except that it requires the request's body hash be included. This is simply a more explicit version of the optional request body digest mode described in RFC 2069.

VSR COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

Related Attacks

SSL/TLS Renegotiation

A serious flaw was recently discovered in SSL and TLS which allows an attacker, positioned on the network in between a client and server, to initiate a protocol renegotiation and inject a single buffer of data into the encrypted stream. In the context of HTTP, this could take the form of a forged HTTP request which can be very serious in certain situations. However, note that this vulnerability does not permit an attacker to decrypt information submitted by either the client or server, unless some additional attacks are conducted which rely on secondary behaviors of either the client or server applications.

In the original paper [ReTLS], two primary attack scenarios were described, one where a client is authenticated using a certificate and the other where clients are authenticated using HTTP session cookies. In the former case, the injected request automatically hijacks the client's identity, since the connection itself is considered trusted by the server and no authentication information need be carried in HTTP headers. However, exploitation of the latter case is more tricky, since a blindly injected request would not carry the necessary HTTP headers to hijack authentication. In this case, an attacker can instead inject an incomplete request which fails to terminate an HTTP header. For example, if an attacker were to inject a partial request which looks like:

```
GET /private/importantAction.cgi?do=evil HTTP/1.1
Host: vulnerable.example.com
X-ignore: 
```

Here, the attacker's final line would not be terminated with a newline. The final character of the injected data would be a space.

Next, the victim's client submits a benign request, perhaps looking like:

```
GET /private/boringPage.html HTTP/1.1
Host: vulnerable.example.com
Cookie: SessionIdentifier=4321873049237492834792384729З
```

Upon receiving these two packets and interpreting them as a continuous stream, the web server would see a single request which looks like:

```
GET /private/importantAction.cgi?do=evil HTTP/1.1
Host: vulnerable.example.com
X-ignore: GET /private/boringPage.html HTTP/1.1
Host: vulnerable.example.com
Cookie: SessionIdentifier=43218730492374928347923847293
```

Here, the web server would ignore the `X-ignore` header (simply because it wouldn't know how to interpret it) and the remaining headers sent by the victim are effectively hijacked for use by the attacker in whatever web application action he chose to perform. Note that most typical session cookies and HTTP basic authentication headers would be prime candidates for hijacking.

In a variant of this attack, Anıl Kurmuş later demonstrated how an attacker could embed a victim's request in the POST parameters of an evil request to gain access to them [TwTLS]. In this attack, the injected data would look something like:

```
POST /sendmessage.cgi HTTP/1.0

recipient=attacker@evil.example.com&message=
```

VSR COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

Here, the victim's request would be appended and treated as form data to be processed by a server's script. In Anıl's example, the popular Twitter website made just such a script available which could provide an attacker access to the POSTed information.

In [ExpTLS], Thierry Zoller provides detailed explanations of several attack variants. In one attack, an injected request would be designed to illicit a redirect code from the server to an unencrypted (non-SSL) web page. If the user does not notice that they are continuing to use the application over an unecrypted link, then this could be an effective attack scenario by forcing all remaining communications to be unprotected using tools such as `sslstrip` [SS].

In the final variant, an attacker injects a full or partial TRACE request which contains a header with JavaScript. Since TRACE responses include submitted headers in the response body, it may be possible to convince some browsers to execute this client-side script after it is returned by the server. In this case, the user's session could be fully hijacked using techniques akin to cross-site scripting exploits.

HTTP Request Smuggling

HTTP request smuggling, as introduced by C. Linhart, et al. in [HRS], is a general class of attacks which is typically used to either poison web caches or hijack user credentials through shared proxies. In the early attack examples, differences in interpretation of HTTP requests are exploited to cause web proxies to interpret received data as one set of requests while web servers behind them interpret the same data as a separate set of requests. The differences of interpretation come about through the use of multiple conflicting `Content-Length` headers as well as exploitation of HTTP parsing bugs in proxies and web servers.

In one relevant example (#5 from [HRS]), an attacker is able to hijack a victim's HTTP basic authentication credentials, provided the victim shares a web proxy with the attacker, by first seeding a partial request with a back-end web server.

For example, an attacker would send two requests back-to-back which look something like:

```
POST /seed.html HTTP/1.0
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: {length of "key=value"}
Content-Length: {length of all data below}

key=valueGET /evil-request.cgi?do-evil=action HTTP/1.0
Content-Length: 0
X-ignore:
```

Here, the attacker's final line would not be terminated with a newline.

In this example, the web proxy accepts the second `Content-Length` header as the valid one and believes it is sending a single request on to the web server. However, the web server believes the first `Content-Length` header is the valid one and responds to the first POST request which is forwarded back to the attacker. The additional data in the stream is then interpreted as a partial second request, but the web proxy is not aware of this. Later, when a victim comes to use the proxy to access the web server, he might send a request which looks like:

```
GET /harmless.html HTTP/1.1
Host: example.com
Authorization: Basic {victim's credentials}
```

But when the web server receives this request (which comes across the same TCP connection used by the proxy) the previously submitted partial request of the attacker is taken to be a prefix of the victim's request. The resulting interpreted request is:

VSR COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

```
GET /evil-request.cgi?do-evil=action HTTP/1.0
Content-Length: 0
X-ignore: GET /harmless.html HTTP/1.1
Host: example.com
Authorization: Basic {victim's credentials}
```

Which of course hijacks the user credentials in much the same way as some TLS renegotiation attacks do.

Over the last several years, VSR has identified custom proxies and applications which suffer from similar vulnerabilities where user-supplied data is used in back-end HTTP requests. In these situations, the simple injection of newlines often leads to a similar result without the need for `Content-Length` manipulation.

For example, suppose a website help library is designed around a front-end script which accepts parameters as follows:

```
/FetchHelp.cgi?Topic=Login&lang=es-es
```

From there, this script looks up the text for the help topic "Login", which is stored in English text. Next, if the user does not desire English, it sends the text on to a third-party translation service to obtain the desired version:

```
POST /translate.cgi?from=en-us&to=es-es HTTP/1.1
Host: translator.example.com
Authorization: Basic {credentials for translation service}
Content-Length: {length of content below}

text={text to be translated}
```

However, since most web frameworks automatically decode URL parameters prior to passing them to custom scripts, the `FetchHelp.cgi` script must be very careful about how user-supplied values are forwarded on in the back-end request. For instance, if an attacker sends a "`lang`" parameter of:

```
lang=es-es%20HTTP/1.1%0d%0aArbitrary-headers-or:%20request%20splitting%0d%0aX-
ignore:
```

Then the resulting back-end request might look like the following if the script failed to validate or re-encode the parameter:

```
POST /translate.cgi?from=en-us&to=es-es HTTP/1.1
Arbitrary-headers-or: request splitting
X-ignore: HTTP/1.1
Host: translator.example.com
Authorization: Basic {credentials for translation service}
Content-Length: {length of content below}

text={text to be translated}
```

While this particular example is admittedly contrived, several instances of this style of header injection have been identified in the past, even though these do not appear to be well documented in publicly available security resources. The result of these vulnerabilities can vary widely, depending on the purpose of the back-end request and the amount of information attackers can obtain on the front-end. In practice, it has been possible in custom proxy scenarios to split HTTP requests and follow up with a second front-end request to obtain results from previous injected requests, provided HTTP sessions are maintained on the back-end.

COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

Mitigation with Digest Authentication

It appears that the attacks described above could be largely addressed through the use of digest authentication. Recall that digest authentication does not permit trivial access to user credentials and it also provides some integrity checking of URIs and the HTTP method.

If a site used digest authentication, an attacker could certainly forge various HTTP headers (if she knew what URI the victim was accessing), but useful attacks would likely be limited to very specific applications and corner cases. For example, consider the `X-ignore` injection initially described in relation to both TLS renegotiation and HTTP request smuggling. Here, an attacker reuses a victim's cookie or basic authentication header to submit her own requests. However, with digest authentication the URI must match the one presented by the victim, which would greatly reduce the kinds of malicious actions an attacker could perform.

In the case of the POST request (Twitter-style) attack, credentials are not reused, but instead divulged through application-specific behaviors. Therefore an attacker could surely obtain the victim's `Authorization` header, but this would then need to be cracked with brute-force or dictionary attacks. Even though digest authentication utilizes a common cryptographic hash (MD5), using precomputation methods (such as Rainbow Tables [RT]) is likely not feasible given the number of site-specific items and nonces which are hashed together. Therefore, only weak user passwords could be targeted.

Provided an HTTPS application contains some page which redirects users to HTTP pages, injection attacks which involve redirecting users to insecure versions of an application would likely work, with some limitations. For instance, a properly configured digest authentication server should not add the insecure version of the site to the list of acceptable domains (originally sent in the `WWW-Authenticate` header), which means a user's browser would not send `Authorization` headers over the HTTP protocol after the redirect. However, with sufficient trickery a victim might be convinced that their session had expired and it is time to reauthenticate. From there, an attacker could conduct Basic authentication downgrade attacks over HTTP, which most users would not notice.

Attacks using injected TRACE requests could be serious, but only if a web server had this feature enabled (which is not advisable for other reasons [XST]) and a user's browser could be convinced to execute the returned malicious script. If this were successful, then an attacker would be able to submit requests from within the user's own browser to perform actions on their behalf.

In summary, use of HTTP digest authentication should mitigate the majority of currently known exploits for HTTP-based TLS renegotiation and some subset of HTTP request smuggling attacks. However, this should not be considered a long-term fix for TLS renegotiation problems, but instead as a stop-gap measure until permanent corrections to SSL/TLS software is widely deployed.

Limitations

One might be convinced at this point that HTTP digest authentication is hands-down a better solution than HTTP basic authentication, and at times, even cookie-based session management. However, the devil is in the details and many common implementations are simply not mature enough for seamless everyday use.

For example, according to [AMAD], Internet Explorer versions 5 and 6 fail to include the URI's query string in the digest hash, which creates incompatibilities and limits the efficacy of URI integrity protection. Apache's `mod_auth_digest`, for its part, has still not implemented several important security features including the `MD5-sess` algorithm, nonce count checking, and `auth-int` support. Mozilla's Firefox also does not appear to support `auth-int` (as of this writing) [FFDA], and neither does Google Chrome [HAHD].

Another serious weakness, common in modern browsers, is that no indication is presented to the user which HTTP authentication method (basic or digest) is being used. As is noted in RFC 2617, man-in-the-middle downgrade attacks are trivially possible (without SSL or other protections) by convincing users to authenticate via basic authentication while sending these credentials to the server via digest authentication. Even with this warning present in the RFC, many browsers (Firefox 3.5.6, Opera 9.62, Google Chrome 3.0.195.38) simply present users with a generic prompt, not indicating which method is in use. (During limited testing, Internet Explorer 8, Opera 10.10, and Safari 4.0.4 at least

VSR COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

added a warning to the prompt when basic authentication over non-SSL connections were used.)

Over a decade has passed since RFC 2617 was released, yet the lack of complete and/or correct support in mainstream browsers and web servers remains a major hinderance for implementors and surely is not encouraging adoption.

Conclusion

HTTP digest authentication should be considered by web application administrators as a temporary mitigation against SSL/TLS renegotiation attacks. However, due to potential incompatibilities, it should be well tested prior to deployment.

In the long run, HTTP digest authentication is not a complete solution for bolstering web security, but as SSL technology shows its age and fragility, web administrators should consider layered approaches to securing sensitive communications.

Acknowledgements

Thanks to George Gal for assistance in testing; Joan Morgan and John Redford who provided helpful comments and suggestions.

VSR

COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.

References

0Prfx Null Prefix Attacks Against SSL/TLS Certificates
http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf

AMAD Apache Module mod_auth_digest
http://httpd.apache.org/docs/2.2/mod/mod_auth_digest.html

DebKeys When Private Keys are Public:
Results from the 2008 Debian OpenSSL Vulnerability
https://cseweb.ucsd.edu/~hovav/papers/yrses09.html

ExpTLS TLS/SSLv3 Renegotiation Vulnerability Explained
http://www.g-sec.lu/practicaltls.pdf

FFDA Mozilla Cross Reference: nsHttpDigestAuth.cpp
http://mxr.mozilla.org/seamonkey/source/netwerk/protocol/http/src/nsHttpDigestAuth.cpp

HAHD http_auth_handler_digest.cc (Google Chrome)
http://src.chromium.org/viewvc/chrome/trunk/src/net/http/http_auth_handler_digest.cc?revision=26723&view=markup

HRS HTTP Request Smuggling
http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

MD5CA MD5 considered harmful today: Creating a rogue CA certificate
http://www.win.tue.nl/hashclash/rogue-ca/

OCSP3 Defeating OCSP With the Character '3'
http://www.thoughtcrime.org/papers/ocsp-attack.pdf

ReTLS Renegotiating TLS
http://extendedsubset.com/Renegotiating_TLS.pdf

RFC2069 An Extension to HTTP : Digest Access Authentication
http://www.ietf.org/rfc/rfc2069.txt

RFC2617 HTTP Authentication: Basic and Digest Access Authentication
http://www.ietf.org/rfc/rfc2617.txt

RT Rainbow Table
http://en.wikipedia.org/wiki/Rainbow_table

SS SSLSTRIP
http://www.thoughtcrime.org/software/sslstrip/

SSLWarn Crying Wolf: An Empirical Study of SSL Warning Effectiveness
http://lorrie.cranor.org/pubs/sslwarnings.pdf

TwTLS TLS Renegotiation Vulnerability (CVE-2009-3555)
http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html

UReTLS Understanding the TLS Renegotiation Attack
http://www.educatedguesswork.org/2009/11/understanding_the_tls_renegoti.html

XST Cross-site Tracing
http://www.owasp.org/index.php/Cross_Site_Tracing

COPYRIGHT © 2010 VIRTUAL SECURITY RESEARCH, LLC
ALL RIGHTS RESERVED.