Weaning the Web off of Session Cookies

Making Digest Authentication Viable

Version 1.0

Timothy D. Morgan

January 26, 2010

Contents

Abstract

In this paper, we compare the security weaknesses and usability limitations of both cookie-based session management and HTTP digest authentication; demonstrating how digest authentication is clearly the more secure system in practice. We propose several small changes in browser behavior and HTTP standards that will make HTTP authentication schemes, such as digest authentication, a viable option in future application development.

Introduction

Early in the history of the world wide web, a transition occurred where web sites began to switch from largely static HTML content to complex, dynamic applications. At the time, authentication of users to an application was supported through HTTP basic authentication, which has very limited flexibility and no security features. As applications progressed and became more dynamic, so did the requirements for user authentication and password management. In response to these forces a relatively new technology, browser cookies, quickly took over as the primary mechanism for tracking application login sessions. Unfortunately, cookies were not originally designed for authentication purposes and cookie standards are both poorly specified and adhered to. For these reasons, the past several years have proven time and again that this approach is riddled with security pitfalls which continue to plague developers and users.

HTTP digest authentication was originally introduced at the same time as HTTP 1.1 and provides marginally more security than the HTTP basic authentication scheme. Even with its limited additional security, we find that it continues to out-perform most cookie-based session management frameworks in the protections it provides. However, digest authentication continues to be largely ignored by application developers since it suffers from the same user interface limitations of basic authentication, preventing developers from controling portions of the end-user experience.

In this paper, we compare the security weaknesses and usability limitations of cookie-based sessions and HTTP digest authentication. In addition, we propose various possible changes in browser behavior and HTTP standards that should make HTTP authentication schemes a viable option in future application development. Note that we do not compare these session management schemes with HTTP basic authentication or SSL/TLS certificate-based authentication for the sake of brevity, as we believe that HTTP basic authentication is clearly a weaker system and that client certificates are clearly stronger from a security perspective.

Cookie-based Session Management

Cookie-based session management is by far the most popular approach to managing user sessions in web applications. Cookies are a general purpose tool supported by nearly all modern browsers and therefore allow for a great deal of flexibility in how user sessions are managed by applications. Here we briefly outline the most common ways session management is currently implemented using cookies.

Users first authenticate using an HTML form, typically with a username and password. (This is why cookie-based session management is commonly referred to as "forms-based" authentication, but the two are not synonymous as we will see later in this paper.) If the user entered correct credentials, then a browser cookie is set (or refreshed) to track the session. Once the user chooses to log out, or their session expires, the session state on the server is deleted and a new, blank session cookie is sent to the user's browser.

The value of the cookie is typically chosen pseudorandomly. Any data associated with the session, such as the current application user and any multi-form state, are stored on the server using the cookie's value as an index. In some less common session management systems, the value of the cookie assigned is actually the session state itself, or a partial session state, which is encrypted using a key known only to the application on the server.

HTTP Digest Authentication

HTTP digest authentication [RFC2617] was designed as an incremental improvement to HTTP basic authentication, which simply sends user credentials in cleartext as a base64 encoded value. HTTP digest authentication was not intended as a generally secure cryptographic protocol and shouldn't be used as such. 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 attacks. 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 generated based on the user password and sent along with several other values. Consequently, passive and active attackers cannot easily take the credentials of a naïve user (who reuses passwords on multiple sites) and use them to 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 flag (`qop=...`) 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 optionally protected.

auth Mode

RFC 2617 introduced revisions to digest authentication that 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.

Comparison

First we list the individual weaknesses of each session management system and then follow with a summary of the differences. Note that throughout this comparison, we assume secure applications will utilize SSL/TLS with a properly authenticated server certificate, since without this, both systems would be insecure in the underlying network stack.

Pitfalls of Cookie-based Sessions

Cookie-based session management is problematic in practice for two primary reasons: it is highly flexible and browser cookies are also used for non-security purposes. Unfortunately, often in computer security, extensive flexibility often becomes it's own pitfall. When presented with highly flexible tools or APIs, some percentage of programmers fail to understand the security ramifications of their implementations and end up shooting themselves in the foot. If instead programmers are provided simple frameworks with few "moving parts", even inexperienced programmers tend to be guided down a secure path which results in fewer security issues. Adding to the flexibility of cookies are market forces which drive browser cookie use, advertising and unauthenticated session management, which is why we continue to see serious flaws in cookie-based session management systems and deployments.

Cryptographic Weakness in Session Identifiers

In order to maintain secure sessions with identified users, the cookies set in browsers must be unpredictable to an attacker. Therefore, it is common practice to generate session cookies with pseudorandom number generators (PRNGs), or to encrypt identifying information (using a key only known to the server) and use the resulting ciphertext as the cookie. However, the typical web developer is rarely an expert in cryptography and seldom has the requisite skills to securely implement basic session management software. Common mistakes in this area include the use of insecure PRNGs, which allow for easily predictable session identifiers, or the incorrect use of encryption algorithms (such as ECB/OFB mode block ciphers or stream ciphers without secondary integrity protection).

Fortunately, over time various web development platforms have built session management frameworks that handle this task for developers, and for this reason, cryptographic weaknesses in session identifiers are now somewhat rare. Note however, these still surface from time to time [PHPWN], and many session management frameworks continue to lack features or default settings which make them secure out of the box.

Session Identifiers in URLs

Browser cookies are commonly used to track users for the purposes of advertising. For this reason, privacy-conscious users often disable cookies for certain sites, restrict cross-domain cookie use, or only accept cookies after a browser explicitly prompts them. To work around situations where a user's browser does not accept or support cookies, many web applications (and in fact some application authentication frameworks) accept session identifiers via URL request parameters.

Unfortunately, this has some serious consequences for the secrecy of session identifiers. When session identifiers are passed in URLs, they are likely to be cached in browser histories, recorded in proxy and web server logs, and worst of all, may be passed to third-party sites in `Referer` (*sic*) headers. Session identifiers in URLs would also be particularly susceptible to cross-site scripting attacks that seek to hijack sessions.

Session Fixation

Since many applications rely on session cookies for purposes other than authentication, application frameworks commonly set session cookies when users first visit a site. Later, when a user authenticates, the user's session is typically "upgraded" to an authenticated state on the server, allowing the user to access restricted areas. However, this sequence of events appears to backward in comparison with typical cryptographic protocols that utilize session keys. That is, session identifiers should be set only after authenticating to the system.

The vulnerability of this behavior was pointed out by Mitja Kolšek in [SFIX] where a few attack scenarios were described. In one example, an attacker walks up to a public terminal and accesses a popular web application, which assigns her a cookie. She records the cookie and walks away. Later, a victim uses the same terminal to access the same website. Once the victim is logged in, the attacker can use the previously recorded session identifier to hijack the victim's account.

More serious attacks are possible if the web application permits users to specify their own session identifiers. For example, if a new session is created and associated with the specified value upon accessing a URL like the following, then additional attack vectors become available:

```
https://example.com/index.cgi?SessionID=12345678
```

It may seem surprising that any web application would allow this, but the issue has surfaced even recently in popular applications [SFOS,SFRT,SFTIM]. In this situation, if an attacker could convince a victim to follow a malicious link to the site and subsequently log in, then hijacking the session becomes trivial.

Lack of Secure Flag

The de facto same origin policy implemented for browser cookies is somewhat limited in that server port and protocol are not taken into account [BSHSOC]. For this reason, if a cookie is set over HTTPS it will also be sent (by default) to the same domain over HTTP for any future requests of that type. Since most HTTPS sites also provide an HTTP interface (even if only to redirect users to the HTTPS version of a site), it is common to see users leaking session cookies over unsecured HTTP communications, particularly if they return multiple times in a given browsing session through bookmarked HTTP links or by typing the site's domain alone into the URL bar.

To change this behavior, the HTTPS site must add the "secure" flag when setting session cookies. This is an easy fix, but it is frequently missed by developers since testing is typically performed over HTTP when the secure flag can't be used. Many standard web frameworks continue to do a poor job of enforcing this setting on HTTPS applications by default, which leaves one more thing for application developers to worry about in each deployment.

Lack of HttpOnly Flag

Since browser cookies are often used in non-security contexts, it is desirable for them to be made available to client-side scripts, such as JavaScript. As one might expect, cookies may be set or read by client-side scripts by default, in stark contrast to HTTP authentication headers, which have typically not been made available. The most common attack that exploits this flaw is cross-site scripting. If an attacker can inject client-side script into a trusted site's page, then session cookies can be accessed and forwarded back to the attacker's server via secondary requests.

In Internet Explorer 6, Microsoft introduced a new cookie flag called `HttpOnly`. When this flag is added to a `Set-Cookie` header, browsers should not make it available to client-side scripts, and instead only allow it to be accessible to the server via HTTP. This brings session cookies back on par with HTTP authentication headers with respect to client-side script limitations, provided developers remember to set it. However, other browsers have been slow to adopt this feature and continue to have difficulties fully enforcing this rule in light of newer features (such as AJAX requests/responses) [HOBS]. In addition, many server-side session management frameworks continue to lack (or only recently gained) support for setting this flag. Because of this slow adoption, it remains more common to find application session cookies lacking this flag than seeing it properly set.

Long-term Session Hijacking

Another weakness exposed by typical cookie-based session management systems is the ability of attackers to hijack sessions in a relatively long-term manner. If an attacker were able to obtain a user's session cookie, either through the weaknesses previously described or perhaps via cross-site scripting or other means, then that session identifier can be used for the duration of the session. In fact, in most cookie-based systems, even if a user were to change their password during a session, the session identifier will not change.

To mitigate this issue, it is a commonly recommended practice to institute two types of session timeout on the server: "soft", and "hard" timeouts. Soft timeouts cause a session to expire if a user hasn't recently accessed an application. Hard timeouts limit the length of a session regardless of when the user last used the active session, and are typically set for something on the order of 1 day. However, many applications only implement soft timeouts. Therefore, once a session identifier is hijacked, an attacker need only to set up a script to repeatedly access the application to keep the identifier alive (assuming the user doesn't explicitly log out).

The core issue here is that once a session is hijacked, there is no continuing or periodic requirement for the user to provide their identity. Some systems attempt to mitigate hijacking by enforcing source IP addresses or by collecting browser-specific information, but much of this is either spoofable or not usable in all situations.

Single Sign-on Weaknesses

It is commonly desirable for multiple applications to support single sign-on (SSO) capabilities. That is, once a user logs in to one web application, access to a specific set of other applications is possible with the same user account without the need to reauthenticate.

Sometimes developers are able to achieve single sign-on by hosting multiple applications under a single domain and allowing session cookies to be sent to every application. However, this simple approach can increase risk associated with cookie exposure. For instance, if one application in the group is vulnerable to cross-site scripting, then all applications could easily be compromised directly. To mitigate this issue, individual applications cookies can be restricted to specific sub-paths with other, less restricted cookie used for cross-application access. However, at this point it is often just as easy to host applications on separate domains.

When single sign-on is required across multiple domains, it is theoretically possible for one application to set a cookie which is also valid in other domains. However, this is now often blocked in browsers by default since the feature has been abused by marketers to track users. Therefore, a common work-around is for the application to generate a cryptographic message, embed it in a URL request or POST parameter, and then redirect the user to a specific page in a second application. The second application decrypts this cookie, determines what user account the user should be assigned, and then sets up a more standard application session for that application.

While many standard frameworks exist for SSO and federated authentication, it is still common to find custom solutions, particularly when separate applications are implemented on different platforms. Custom solutions can be particularly vulnerable, since the cryptographic messages are often exposed in URLs and any weakness in the cryptographic implementation (such as those mentioned previously in the context of session identifier generation) may permit spoofing or modification of these messages.

Limitations of Digest Authentication

HTTP digest authentication is a major improvement over HTTP basic authentication, but it still suffers from several user interface limitations that prevent its use in the majority of applications.

Lack of Customizable Login Form

A major limitation of HTTP digest authentication, and HTTP authentication schemes generally, is that there is no obvious way for application developers to customize the login process. Developers are accustomed to having (and

businesses demand) a way to provide users with a custom HTML login page for a user-friendly experience. HTTP authentication methods are, however, based on received HTTP response types and headers instead of HTML forms which makes them difficult to customize. This is a primary reason why few end-user applications rely on any form of HTTP authentication.

Lack of Application-Driven Logout

Perhaps a larger flaw in HTTP authentication schemes is the lack (or perceived lack) of a server-driven log out capability. Users are conditioned to expect a log out button in secure web applications which causes their session to be expired. With HTTP authentication methods, many browsers do provide features for clearing cached HTTP authentication credentials, but each browser interface is different and it cannot be directly driven by the application. For instance, in cases where a user's session has expired, there is no easy way for applications to signal to a browser that the cached credentials should be cleared. Some applications attempt to achieve this by sending users an HTTP 401 response, but this causes almost all browsers to prompt the user for another password without allowing for a log out condition.

Lack of Choice in Hash Storage Format

Because of the way digest authentication works, servers must store password hashes (A1 values) in the format used by the protocol, which is roughly:

```
MD5(USERNAME + ":" + REALM + ":" + PASSWORD)
```

This hashing format is reasonably secure against precomputation attacks, provided that `USERNAME`s and `REALM`s are not commonly reused. However, if additional digest authentication hash algorithms are introduced in the future (such as SHA-256 or Whirlpool), then all user passwords would need to be reset to support the newer hash format.

Potential Exposure of Weak Passwords

If an attacker were able to steal a user's digest authentication response hash, they may not be able to reuse the hash with the server, but it would be possible to crack the user's password if they chose a weak one. This is because all of the information (nonces, username, etc) are included in plaintext in the response header. Note that precomputation attacks, such as the use of Rainbow Tables, would almost certainly not be possible due to the storage and computation requirements introducted with multiple nonces and counters, but poorly chosen passwords could be cracked directly using dictionary attacks.

Comparison Summary

HTTP digest authentication lacks many of the security weaknesses of commonly used cookie-based session management. Since the cryptographic negotiation is standardized there is less room for cryptographic weaknesses. For instance, even if both a server and client were to select predictable nonces, the response hashes would be at best vulnerable to replay attacks, but would not be predictable outright. In addition, since the protocol is standardized and not used for tracking users, there is no risk of leakage through URLs, as session identifiers sometimes are.

Session fixation is also not an issue with digest authentication, since the very first request tied to a session is authenticated. The potential weakness created by the lack of a secure flag with session cookies is unlikely to occur in digest authentication unless application developers or web administrators go out of their way to add a non-HTTPS URL to the protection space advertised by the server.

The need to add the `HttpOnly` flag to cookies is also irrelevant with digest authentication, since browsers do not provide access to these headers in client-side scripts without additional weaknesses (such as the TRACE or TRACK methods being enabled). Even if a digest authentication header were hijacked through some means, it would likely not be usable to hijack a user session directly, since nonce counts would change and server nonces themselves may

change periodically. This is a significant advantage over session cookies.

Digest authentication also provides a form of built-in single sign-on by allowing a server to specify other servers in the same protection space. The servers in this space would not need to coordinate any session information on the back-end, but would merely need to share the user's A1 hash. Additionally, private session information could be communicated between servers through encrypted nonces and "opaque" data which would be forwarded by clients automatically.

Additional security features of digest authentication include some level of integrity protection in light of limited injection attacks (such as TLS renegotiation and HTTP request smuggling [HDI]) and optional mutual authentication. However, some of these features are highly implementation dependent.

It is worth noting that digest authentication does currently rely on the MD5 hash algorithm which has been widely demonstrated to be flawed for certain cryptographic properties. However, the primary property digest authentication relies on is preimage resistance. For instance, so long as the legacy RFC 2069 mode is avoided, clients will provide a nonce as a part of the response hash which should make chosen plaintext attacks difficult for malicious servers. In addition, digest authentication does leave open the possibility of other hash algorithms to be added in the future.

The most serious security weakness of digest authentication, when compared to standard cookie-based session management, is that weak user passwords could be cracked if authorization headers were ever obtained by an attacker. However, given the fact that these headers are much less likely to be stolen (when compared to cookies) and cannot be reused to hijack sessions immediately, this seems like a good trade-off. To mitigate this problem, users should be required to select complex passwords, which is considered a security best practice anyway.

Possible Solutions

The reason digest authentication hasn't caught on has little to do with the security of the protocol. The primary reason is the lack of a flexible user interface. If this barrier to entry could be removed, it should be much easier to convince application developers and administrators of its security virtues.

Form-based HTTP Authentication

Over the years, various web developers have published methods which attempt to provide customizable HTML forms for login with HTTP basic and/or digest authentication [HAHF,SBHA]. The most promising recent method is described in [PLP] by Berend de Boer where AJAX functionality is used to take control of the authentication process. The approach is outlined in the following JavaScript-like pseudocode:

```
XHR = {{new XMLHTTPRequest Object}}
username = {{username from HTML form}}
password = {{password from HTML form}}

XHR.open("GET", "/private/login-check.cgi", false, username, password)
XHR.send()

if(XHR.status == "200")
  redirect("/private/landing.cgi")
else
  reportError("Login Failed")
```

Here we assume that the page contains an HTML form with standard username and password fields. Upon submitting the form, JavaScript executes the routine listed above. The username and password are supplied to the browser's `XMLHttpRequest` open method, which means the browser can use them for basic, digest, or any other supported authentication method. If the login was successful, the AJAX request will see a 200 status code returned in HTTP headers and JavaScript can then redirect the user to a landing page. If the login fails, the script will (in theory) report a custom error to the user.

Unfortunately, this approach doesn't quite work without a small adjustment. Upon receiving a 401 error code from the server, *even in response to an AJAX request*, the majority of popular browsers will immediately prompt users for their credentials with the default popup. Therefore, if users fail to type the correct password the first time, the rest of the authentication process is taken out of the web developer's hands. To work around this problem, site developers could design the `/private/login-check.cgi` script to return a different kind of error code (perhaps a different 400 code) which would be ignored by the browser and could allow continued processing in JavaScript.

This work around skirts the HTTP specification and is not particularly convenient, but fortunately there may be hope for a correction to this browser behavior. The recent W3C working draft [XHR] for the `XMLHttpRequest` standardization effort contains the following:

> `If authentication fails, and request username and request password are provided, user agents must not prompt the end user for credentials. [RFC2617]`
>
> ***NOTE: End users are not prompted if credentials are provided through the open() API so that authors can implement their own user interface.***

It seems someone in the standards community has already recognized this shortcoming. Assuming this browser behavior is changed, implementation of custom login forms for any HTTP authentication method can be made quite simple. In fact, login forms could be entirely static (that is, with no dynamic server-side scripts required) and if some browsers do not fully support the JavaScript mechanisms driving the form, a link can be provided to the authenticated area where the traditional HTTP authentication prompt would be triggered.

Approaches for Logout

Perhaps a more serious limitation of HTTP authentication in browsers is the lack of an application-induced log out. A number of browser-specific hacks have been described [CACS] that achieve this in JavaScript. For instance, in the case of Internet Explorer, support is provided for a proprietary command designed specifically for this task [CACC]. A similar result can be achieved in Firefox through the use of an asynchronous `XMLHttpRequest` where invalid credentials are provided to the `open()` method which causes the old credentials to be forgotten. (In order to avoid having users prompted upon receiving the 401 from the server, the request is immediately aborted before a response can be processed.) Other browsers may provide other workarounds, but these approaches are clearly less than ideal.

One might wonder why a standard response code or header was not added to HTTP that induces a log out behavior. As it turns out, many within the HTTP standards community believe the capability to provide a log out already exists, it's just that browser implementations have let us down. Indeed, there is nothing stopping browser vendors from implementing user interfaces that provide a menu item or other button for clearing site credentials (and many modern browsers do implement this), but the key to developer adoption is giving web applications a way to force a log out. However, even this can easily be provided with a simple change to how 401 responses are processed in browsers. Currently, anytime a popular browser receives a 401 response, it immediately prompts the user for credentials regardless of the user's previous login state. Instead of doing this, it is suggested the following check could be performed:

```
if cached credentials exist for (auth_scheme, realm, site):
  Clear cached credentials
  Display body of 401 response
else:
  Prompt user for new credentials
```

Here, the `auth_scheme` is the type of HTTP authentication (basic, digest, etc) and the `realm` is the realm advertised in the `WWW-Authenticate` header presented in the initial login and again during log outs. The `site` is a same-origin designation which may vary from one authentication scheme to the next, but is typically just the website that originally prompted the user for credentials (and is now concluding the login session). This same-origin designation would be somewhat more complex in single sign-on scenarios with digest authentication, but current 401 response credential clearing rules could be used.

There is nothing in the HTTP specification to prevent this more intelligent 401 response processing behavior. By displaying the body of a 401 page during the log out, application developers can fully control the logout landing page and provide links to public areas or back to the login page. This seems to be the simplest way to achieve a log out, as no JavaScript hacks or additional standards need to be defined.

However, it may be that any given browser vendor is afraid to change 401 response handling because so many other browsers already behave this way, or because application developers are accustomed to it. In order to nudge browser vendors in the right direction, it may be necessary to make a small change to HTTP authentication standards. One approach would be to add a new HTTP header, `Authentication-Control`. This header would be optionally included in 2XX and possibly 3XX responses to trigger changes in authentication state with a client. We propose the header be generalized and flexible to allow future (potentially very sophisticated) authentication protocols to utilize it. The basic grammar might look like:

```
AuthenticationControl = "Authentication-Control" ":" auth-info
  auth-info           = auth-scheme 1*SP realm "," #(auth-param)
  realm               = "realm" "=" quoted-string
  auth-param          = token "=" ( token | quoted-string )
```

Here, we simply require the authentication scheme and the realm. Any additional name-value pairs would be scheme-specific. For instance, if a future authentication scheme wished to define a name-value pair to change a cryptographic key for an existing session, it could easily do so in any response.

For our purposes, we want the ability to terminate digest authentication sessions. Using this new header a response

for a log out request might look like:

```
HTTP/1.1 200 OK
Date: Thu, 15 Jan 2010 00:00:00 GMT
Authentication-Control: Digest realm="My Auth Realm", terminate="true"
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 125

<html><body>
<h1>You've Been Logged Out</h1>
<a href="http://example.com/login">Click Here To Log Back In</a>
</body></html>
```

Note that if a header like this were introduced in a new RFC, legacy browsers would simply ignore it. To manage the transition period, a log out response page could include JavaScript workarounds for major browsers to accomplish the same goal.

While the addition of an HTTP header might seem overkill given how browsers could address this specific issue with no standards change, the suggested HTTP header has the potential benefit of added flexibility for all HTTP authentication schemes.

Practical Concerns

Immature Digest Implementations

While the lack of flexible HTTP authentication logins and logouts is the primary hurdle for adoption of digest authentication, other issues exist in common implementations of the protocol itself. 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 HTTP server (`mod_auth_digest`) and Tomcat have still not implemented several important security features including the `MD5-sess` algorithm, nonce count checking, and `auth-int` support [AMAD,ATRB]. Mozilla's Firefox and Google Chrome do not appear to support `auth-int` (as of this writing) [FFDA,HAHD].

Weak User Interfaces for HTTP Authentication

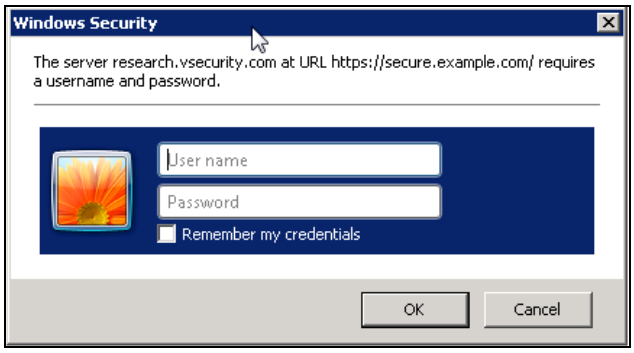
A number of serious weaknesses exist in user interfaces for HTTP authentication in popular browsers that should be addressed. These include trivial man-in-the-middle downgrade attacks with basic authentication and phishing attacks through cross-domain page resources.

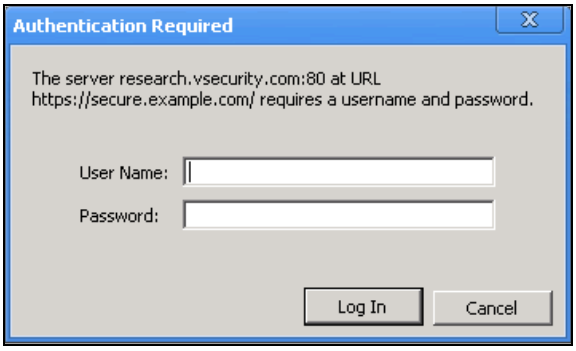
As is noted in RFC 2617, man-in-the-middle downgrade attacks on digest authentication are trivially possible (without SSL or other protections) by convincing users to authenticate via basic authentication while sending these credentials to the server via digest authentication. Even with this warning 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 added a warning about insecure submission to the prompt when basic authentication over non-SSL connections were used. However, even these warnings are somewhat limited in that they do not describe precisely which authentication protocol is being used.

Exacerbating this issue is the fact that browser password managers do not differentiate between credentials cached for different HTTP authentication types. If at one point in time a user authenticates to a web application using HTTP digest authentication and saves those credentials, all tested browsers will happily reuse those same credentials to prefill a login form for basic authentication to the same server and realm. Safari (version 4.0.4 under Windows) actually goes a step further and uses previously cached digest credentials for basic authentication without prompting. While this should certainly be addressed, it is likely that applications will primarily utilize customized login forms in the future. For this reason, browsers should consider implementing controls for disabling basic authentication on user-specified sites or automatically for sites which have previously provided more secure authentication schemes. At a minimum, cached credentials must be differentiated based on HTTP authentication scheme and only submitted to the same or more secure schemes.

Another classic problem with HTTP authentication prompts is that they can be triggered in a confusing way through page subelements [BSHHA]. Consider a message board application hosted on `https://secure.example.com/` where users may include images from third-party web sites. An attacker could post a message containing an image URL of `https://research.vsecurity.com/evil.png`. Once posted, the attacker could then configure his web server to require HTTP basic authentication. Later, when users view the message posting they would be prompted for a password. This may be confusing for users, since the web site they are currently viewing is the message board, but the credentials are being requested by the attacker's site.

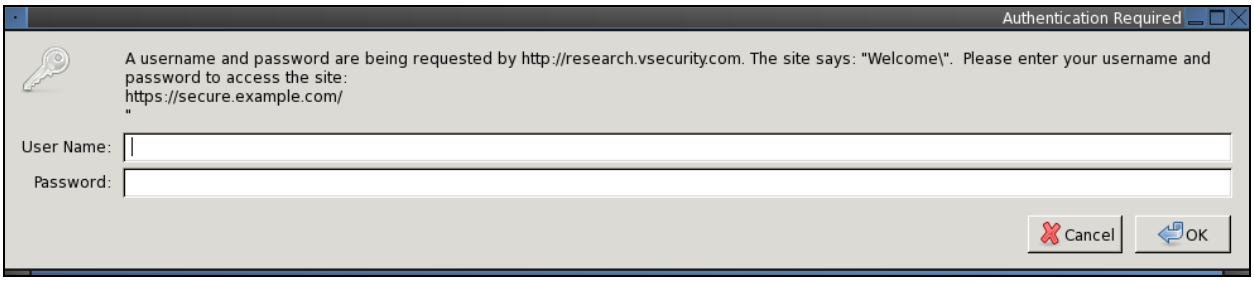
This would not be nearly as serious of an issue if browsers made it more clear in the user interface which web site was requesting the credentials. While all browsers tested did include the domain and possibly the protocol in the authentication pop-up, this information is commonly included in a verbose text description along with the site's realm. For instance, with Internet Explorer 8 and Chrome 3.0.195.38 under Windows 7 an attacker could easily use a realm to confuse users further:





In both of these examples, the realm was "`URL https://secure.example.com/`", which is designed to hopefully confuse users into thinking that was the site they were authenticating to.

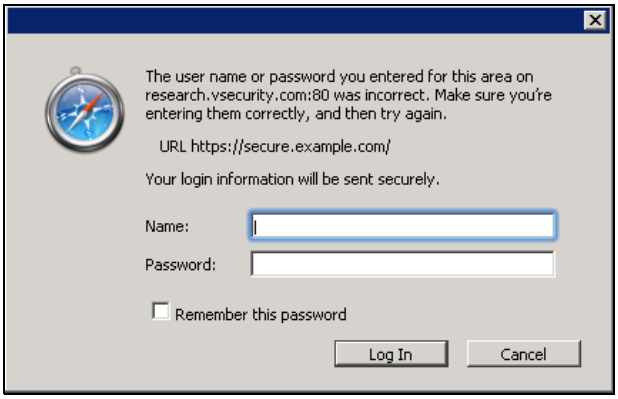
Firefox 3.5.6 under Linux did a slightly better job of differentiating the realm by enclosing it in quotes ("), but this could still be confusing:



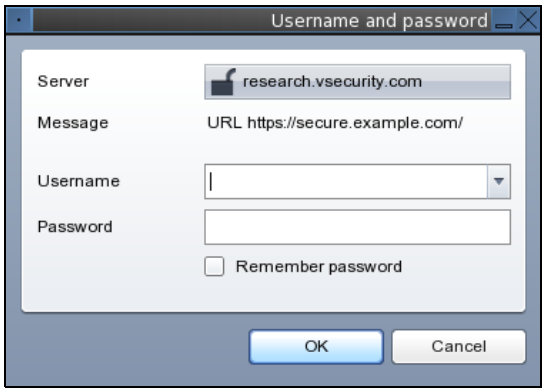
In this case, the malicious realm was approximately the following (with a large number of trailing tab characters):

```
Welcome\". Please enter your username and password to access the site:
https://secure.example.com
```

Safari 4.0.4 under Windows also attempts to differentiate the realm from the site, but it would likely still be easy to confuse many users:



Opera 10.10 under Linux did much better at making these fields clearly separate by highlighting the domain next to a label:



In summary, we recommend that browsers implement the following controls in relation to HTTP authentication:

- Clearly separate the domain and realm in the authentication dialog.
- If an HTTP authentication scheme is being used which is considered insecure and SSL/TLS is not in use, include a clear warning message in the dialog.
- Consider adding a special warning to the dialog if it is for a cross-domain request.
- Provide a button or similar construct which launches a secondary window fully describing the type of authentication the server is requesting, including:
  - Authentication scheme chosen by the browser
  - Secondary authentication schemes offered by the server (a single 401 response may advertise multiple)
  - Listing of full URLs protected (digest authentication may have multiple)
  - Realm
  - Information on SSL/TLS, if applicable (perhaps in separate dialog)
- Add dialogs in the browser settings area to show users which temporary credentials are currently in use, what authentication schemes they utilize, and perhaps even a short history of when the credentials have been used in the past.
- Provide a way to disable specific authentication schemes on a per-site basis.
  - If given site advertises the digest scheme (or a more secure scheme), then basic authentication should be automatically disabled for that site in the future (with user intervention possible).
  - Eventually, basic authentication should be disabled for all sites by default with an exception mechanism added for backward compatibility.
- Password managers must differentiate between different authentication schemes, disallowing use of passwords for less secure schemes.

Application Server Support

Once flexible HTTP authentication support makes its way into browsers and popular digest authentication implementations are corrected, many application developers will continue to be hesitant to adopt digest authentication if session management frameworks do not begin supporting it. Application developers are currently accustomed to managing user sessions directly in application code, but today most server-side implementations of digest authentication exist in web servers, not application servers. To truly catch on as a usable alternative, a transition of implementations to application servers will likely be necessary.

Conclusion

We have shown in this paper how cookie-based session management systems are highly fragile. One might argue that the easiest way forward is to standardize cookie-based session management in a secure way. However, as one looks at the cryptography involved in implementing secure password protocols, complex client-side logic would be required. This is not the kind of logic which could be easily (or safely) implemented in JavaScript. One could add standardized JavaScript features to browsers which hand off these complex interactions in new cookie-based protocols, but this would be an exercise in reinventing the wheel. Besides digest authentication, several other HTTP authentication schemes exist or are in the development process [MAPH,RFC4169,RFC4559]. For this reason, we think a much better approach would be to make HTTP authentication schemes widely usable now and in the future. Since digest authentication is already widely implemented and provides security benefits over the status quo, we believe it is a great first step.

Acknowledgements

Thanks go to the following for providing helpful suggestions: Jan Algermissen, George D. Gal, Dan Kaminsky, Jason Morgan, Joan Morgan, and Yutaka Oiwa.

References

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

ATRB Apache Tomcat Subversion Respository: RealmBase.java
http://svn.apache.org/repos/asf/tomcat/trunk/java/org/apache/catalina/realm/RealmBase.java

BSHHA Browser Security Handbook: HTTP Authentication
http://code.google.com/p/browsersec/wiki/Part3#HTTP_authentication

BSHSOC Browser Security Handbook: Same-Origin Policy for Cookies
http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_cookies

CACC ClearAuthenticationCache Command
http://msdn.microsoft.com/en-us/library/ms536979%28VS.85%29.aspx

CACS Cookieless Authentication - Client side
http://www.nanodocumet.com/?p=6

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

HAHF HTTP Authentication with HTML Forms
http://www.peej.co.uk/articles/http-auth-with-html-forms.html

HDI HTTP Digest Integrity: Another Look, in Light of Recent Attacks
http://www.vsecurity.com/download/papers/HTTPDigestIntegrity.pdf

HOBS HTTPOnly: Browsers Supporting HTTPOnly
http://www.owasp.org/index.php/HTTPOnly#Browsers_Supporting_HTTPOnly

MAPH Mutual Authentication Protocol for HTTP
https://www.rcis.aist.go.jp/special/MutualAuth/

PHPWN phpwn: Attack on PHP sessions and random numbers
http://samy.pl/phpwn/

PLP REST Based Authentication: Personalised login page
http://www.berenddeboer.net/rest/authentication.html#login-page

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

RFC4169 Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA) Version-2
http://www.ietf.org/rfc/rfc4169.txt

RFC4559 SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows
http://www.ietf.org/rfc/rfc4559.txt

SBHA HTTP Authentication
https://wiki.slugbug.org.uk/HTTP_Authentication

SFIX Session Fixation Vulnerability in Web-based Applications
http://www.acros.si/papers/session_fixation.pdf

SFOS osCommerce 'oscid' Session Fixation Vulnerability
http://www.securityfocus.com/bid/34348

SFRT RT Session Fixation Vulnerability
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3585

SFTIM IBM Tivoli Identity Manager Session Fixation Vulnerability
http://www.securityfocus.com/bid/35779

XHR XMLHttpRequest: W3C Working Draft 19 November 2009
http://www.w3.org/TR/XMLHttpRequest/#the-send-method