Secure Session Management With Cookies for Web Applications

Chris Palmer <chris@isecpartners.com>

iSEC Partners, Inc
444 Spear Street, Suite 105
San Francisco, CA 94105
https://www.isecpartners.com/

Version 1.1

September 10, 2008

1 Introduction

Strong session management is a key part of a secure web application. Since HTTP does not directly provide a session abstraction, application and framework developers must bake their own using cookies.[1] In this article I am to help developers aovid the common pitfalls that result in unsafe applications.

Developing an application with secure session management requires developers to understand a few crucial subtleties of cookies — their attributes, their values, and how to keep them confidential — and to understand how real-world attackers are abusing weak session management in real applications today.

Unfortunately, it is surprisingly easy to make a mistake, even when the application uses a sophisticated application framework such as .NET or or J2EE. These frameworks provide session management abstraction layers that hide some of the details of session management from the application's developers. That's good, as far as it goes. However it isn't enough, because not all applications have the same security requirements. Often, developers misunderstand the services provided by the frameworks, or session security in general, and this can lead to severe security issues for the application.

For example, many application designers and developers understand the need to use TLS/SSL[2] when the user is logging into a sensitive application: it's important to send the user's credentials to the *true* application server, not an impostor; to encrypt the data to protect them from the prying eyes of network eavesdroppers; and to ensure that the request was not tampered with in transit. But consider the server's response: it sends the client a cookie, by which it recognizes the client on subsequent requests, providing continuity of the now-authenticated user's session. That makes the cookie equivalent to a password during the time the session is valid: after all, it is the sole token by which the server authenticates the user after the first login request. (This is also why keeping the session validity window short reduces the risk from some types of session hijacking threats.)

[1] Although it is possible to put a session identifier or session state in a query parameter, doing so may compromise the security of your users' sessions. See Appendix C for more information. If much of the content in this article is new to you, you should read the appendices last.

[2] For the rest of this document, I use the term "TLS" to refer to TLS 1.0/SSL 3.0 or greater.

Because session cookies allow access to the application, like a short-lived password, their exposure is a big risk and protection is important. If the cookie is exposed over a plaintext HTTP connection or to an impostor server, the user's account is subject to immediate compromise by a network attacker! Yet many applications either use HTTP for post-authentication flows, or allow an attacker to force the exposure of the cookie over HTTP (or possibly to a malicious server).

In addition, users must be able to request the login form over an authenticated channel, i.e. HTTPS. Users have no reason to trust that a login form they retrieve over HTTP is the true login form that will post to the true application. Security-conscious web applications often refuse to work at all over HTTP, redirecting users to the secure version and reminding them to check for the TLS indicators in the browser chrome (e.g. the lock icon). The better online banks do this, for example.

Fortunately, we can achieve strong session management with a bit of care and a good understanding of how browsers interpret cookies.

2 Cookie Review

Most web application frameworks use *client-side cookies* to index a *state table* on the server side. Session state is usually represented with a special-purpose object type, stored on the server, and could contain anything relevant to the application:

- user profile,
- user privileges,
- cached data from a back-end store,
- browsing history and page flow state, or
- CSRF prevention tokens.[3]

Sometimes applications try to be "stateless", meaning that the server does not store session state. Instead, these applications store the session on the client side within the cookie or page body. There are some additional security considerations which application developers must take into account when developing stateless session management systems. The applicability of these issues depends highly on the application's requirements and implementation. Consider these potential issues:

- Unless the cookie's confidentiality is protected, attackers can read the session state. This could lead to any number of privilege escalation, data integrity, or information leakage issues.
- Unless the cookie's integrity is protected, attackers can write the session state. This also could lead to any number of privilege escalation, data integrity, or information leakage issues.
- Cookies are limited in size, decreasing the amount of storage available for session state.[4]

[3] Cross-site request forgery (CSRF) is another common and potentially severe session management vulnerability. I don't address it in this article because it is already addressed well in Jesse Burns' whitepaper, available at https://www.isecpartners.com/files/CSRF_Paper.pdf.

[4] RFC 2965 (http://www.faqs.org/rfcs/rfc2965.html) says that "general-use user agents SHOULD provide each of the following minimum capabilities: [...] at least 4096 bytes per cookie [...]." While some clients may accept larger cookies, it may be unwise to depend on that behavior. (Mobile clients are particularly bad about supporting the full 4096 bytes.) Note also that this RFC is inaccurate in places, with discrepancies between how browsers and servers actually behave and what the RFC says they should do. As always where there are discrepancies, trust only in the least common denominator of functionality in the common browsers.

- The greater the ratio of cookie size to average resource size, the more bandwidth-intensive this mechanism is. (Compare this to the trivial 32 bytes of a hex-encoded 16-byte session ID.)

- Since the client has complete state, it may be possible for a malicious user to execute replay attacks. Consider a cookie that contains authorization for a given action: The server may not have a way to know not to honor it again, or in a different context.

- Invalidating an active session becomes more difficult because the server no longer holds the state material. While the server can set an invalid session cookie in the client to indicate session closure, if an old cookie is discovered and used by an attacker the server will have no way to know not to honor it.

Despite these considerations, there are compelling benefits to storing the session state on the client side, including:

- Simplified load-balancing and high availability since clients can be directed to any web server supporting the application. If a given member of the web server cluster crashes, the sessions it was running are not lost.

- Reduced server memory footprint by removing the requirement to store complete state for each client.

- The potential for lower response latency and lower equipment costs, due to less need for dedicated load-balancing and session-persistence systems.

- Overall reduced deployment complexity.

For a discussion on how to implement client-side session storage as prudently as possible, see Appendix D.

2.1 Cookie Contents and Attributes

In addition to the name and value attributes, the server can specify several attributes for a cookie which affect how the browser will use it. Attributes can affect how long the browser stores the cookie, if it persistently stores the cookie at all, whether the browser will send it over any connection or only over HTTPS connections, and what server hostnames it will send the cookie back to, among other things.

To set a cookie, the server puts a Set-Cookie HTTP header in the HTTP response. For review, this is shown in Figure 1.

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=3E880015CF879C5014FEAB04C6623203; Path=/myapp
Content-Type: text/html;charset=ISO-8859-1
Content-length: 5219
...Possibly other headers...

...Response data here...
```

Figure 1: A web server setting a cookie for the client.

The client sends the cookie back to the server by putting a Cookie header in the request, as seen in Figure 2. Note that the client only sends the name = value pair(s), but not the cookie attributes. This is because the attributes are instructions from the server to the browser about how and when to send the cookie back.

This list summarizes the atttributes that may be set on a cookie. These attributes are declared by the server when setting the value of the cookie on the client.

***Path* (string: path prefix for URI)** is a namespace management mechanism. The browser will only send the cookie

```
GET http://www.example.com/myapp/index.html HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=3E880015CF879C5014FEAB04C6623203
...Possibly other headers...
```

Figure 2: The client sending the cookie back to the server in a request.

when it is requesting pages underneath the resource hierarchy named by *Path*. For *Path* = "/stuff", the browser will send the cookie when requesting /stuff/, /stuff/gadgets.html and /stuff/things/goodies.html, but not when requesting /doodads.html. If there is no *Path* attribute, the browser behaves as if it had been set to "/".

***Expires* (string: date)** tells the browser to store the cookie in a persistent store (the "cookie jar") on the client machine and to send it back to the server until the given time. If no *Expires* attribute is given, the browser will discard the cookie when it exits.

***Domain* (string: host or domain name)** tells the browser what hostnames to send the cookie back to. For *Domain* = "www.example.com" the browser will send the cookie to hosts whose names exactly match "www.example.com", while for *Domain* = ".example.com" or *Domain* = "example.com" the browser will send the cookie to "www.example.com" and "wiki.example.com". If *Domain* is omitted or blank, the browser will only send the cookie to the exact hostname from which the cookie was set. Omitting *Domain* is often the most secure choice for this reason. It is also often sufficient to set *Domain* to a third-level domain name, such as "www.example.com".

For more on how browsers treat the *Domain* attribute, see Appendix E.

***Secure* (boolean)** tells the browser to send the cookie to a server only over a 'secure' (HTTPS) connection. If omitted, the browser will send the cookie over any type of connection. Without this flag, the browser doesn't know that the cookie is sensitive and in need of protection.

***HttpOnly* (boolean)** tells the browser not to let JavaScript read the cookie value, such as via *document.cookie* (although it may still write or append to the cookie!). *HttpOnly* was originally only supported in Internet Explorer, but now works in Firefox as well.

I have put together two demos of browser behavior with respect to cookie attributes: https://labs.isecpartners.com/chris/cookie-test/ and https://labs.isecpartners.com/chris/httponly/.[5] Use a proxy like WebScarab[6] to watch what cookies your browser sends and receives in each request.

The attributes most important to the security of a cookie are *Secure* and *Domain*: they tell the browser *how* to send the cookie *to whom*.

Path is Not a Security Boundary

While it might seem that *Path* should be a security boundary — i.e. that applications accessed via different paths on the same server should not be able to set or get each other's cookies — in fact it is not.

The reason is that the same-origin policy applies only to the domain name, not to the path. Although *document.cookie* in the context of a page from either application will not contain the other's cookies, a malicious application can craft a page that contains an iframe which in turn contains a page from the other application. Script in the enclosing page can then access the other application's cookie. The browser allows this because both applications are in the same domain.

[5] The code for these pages is reproduced in Appendix E.

[6] Available at http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.

HttpOnly[7] can help provide some marginal defense against cross-site scripting (XSS) attacks. I recommend that you use it where possible (i.e. whenever the client-side portion of the application does not need read access to the cookie), but that you don't depend on it as your sole means of XSS defense. When an XSS vulnerability is present in your application, an attacker can still do plenty of damage without reading a user's session cookie.

3 Attack Classes

Attackers want to know the value of the session cookie (or other sensitive cookies, if any). They can do so by *guessing* it, *discovering* it, or *setting* it.

If the attacker succeeds by any of these means, it's game over: they can hijack the victim's session and compromise the user's account and any sensitive data it holds. The attacker does this by inserting the stolen session cookie into their browser and making requests to the application. Since authentication is based solely on the cookie and HTTP is a stateless protocol, the server is no longer able to tell the difference between the attacker and the victim — i.e., the attacker has stolen the victim's session.

3.1 Cookie Guessing Attacks

If the session ID is low in entropy (say, 32 bits or less), the attacker can guess IDs by brute force.

Consider a scenario in which an attacker, Mallory, wants to guess the session IDs used by an application. The application uses incrementing session IDs, which have essentially zero entropy — i.e., they are completely predictable. The application gives out session IDs starting at some number and simply increments that number for each new session.[8] Mallory doesn't yet know this about the behavior of the application, but she can discover it and abuse it.

Mallory registers for an account and then logs in. She notes that her session ID, stored in a cookie, is 10092. To test her hypothesis that the session IDs are low in entropy, she logs out and logs in again. This time, her session ID is 10117. From this, Mallory can surmise two things: the session IDs are probably small (these two happen to be 14-bit numbers) and they are probably sequential (because they are very close together and apparently increasing). By taking more samples, Mallory can confirm her conclusion.

If she simply sets her session cookie to 10116, chances are good she'll suddenly be "authenticated" as some hapless victim! She will not necessarily be able to attack a specific user for any given attempted attack, whether it's Alice, Bob, Carol, or Dave; but she has a very good chance to steal the session of any one of them. Over time, she has a good chance to attack a specific user. The more users that are logged in at any one time, the greater Mallory's chance of hijacking one of their sessions is.

[7] See http://msdn2.microsoft.com/en-us/library/ms533046.aspx.

[8] For example, when using databases with auto-incrementing integers as primary keys, and using the primary key of the session table as the session cookie.

Entropy vs. "Randomness"

Even pseudo-random 32-bit numbers, such as those generated by the *rand* C library function (and functions in higher-level languages that use it[a]) are not safe. For one thing, even if you used 32 bits of real entropy, Mallory can simply start brute-force guessing. The larger your average number of concurrent users (relative to the size of the session ID), the better probability Mallory has of hijacking somebody's session on each attempt.

But even if the application used 128-bit pseudo-random numbers, she could still starting guessing the next session ID in the PRNG's period through a technique called *lattice reduction*.[b] We therefore need to generate session identifiers that are both *large* (128 bits is sufficient) and *entropic*. Nothing else will do.

To identify low-entropy session identifiers in your application (using black-box techniques), check out Michal Zalewski's stompy[c] tool and the SessionID Analysis feature in WebScarab.

[a] The Mersenne Twister is a particularly good pseudo-random number generator (PRNG). But beware: "good" for a PRNG means only that its outputs are *evenly distributed* in its range, with a low likelihood of repeating a given output before its period starts over — and yes, these functions are periodic, hence deterministic.

[b] For example, see "How to Crack a Linear Congruential Generator" (http://www.reteam.org/papers/e59.pdf) by Haldir.

[c] See http://lcamtuf.coredump.cx/stompy.tgz and http://csrc.nist.gov/rng/.

Some web application frameworks, such as .NET and good implementations of J2EE, use high-entropy session IDs. But many (or even most) other web app frameworks do not. Recent versions of PHP can have large, entropic session IDs, although this security-critical feature is disabled by default.[9]

If you're not using .NET or Java, have a look at your framework's source code. It's pretty easy to quickly identify weak, low-entropy session ID generation in the code, because they use things like:

- the time and date,
- a 'random' static string in the source code,
- the output of C library *rand*,
- the output of *java.util.Random*,
- small (32 bits or less) numbers, or
- a cryptographic hash (like MD5) of anything low in entropy

to generate their session IDs. Conversely, high-entropy session ID generators use things like:

- *java.security.SecureRandom* (Java),
- *System.Security.Cryptography.RNGCryptoServiceProvider* (.NET),
- /dev/urandom, /dev/(s)random (if the latter, look for exhaustion attacks!),
- OpenSSL's *RAND_bytes*,[10] or
- a hardware security module.

[9] As of 25 April 2008, http://us2.php.net/manual/en/session.configuration.php says: "*session.entropy_length* specifies the number of bytes which will be read from the file specified above [the *session.entropy_file*, such as /dev/random]. Defaults to 0 (disabled)." It should be 16 (= 128 bits).

[10] But note that *RAND_bytes* can degrade to a PRNG on some platforms.

Fortunately, getting entropic values is easy. See Figure 3 and Figure 4 for simple examples.

```
import java.security.SecureRandom;
import java.math.BigInteger;

public class EntropicToken {
    public static TOKEN_SIZE = 16;    // 128 bits
    private static SecureRandom SECURE_RANDOM = new SecureRandom();

    byte [] token;

    public EntropicToken() {
        token = new byte[TOKEN_SIZE];
        SECURE_RANDOM.nextBytes(token);
    }

    public String toString() {
        return (new BigInteger(token)).toString(16);
    }
}
```

Figure 3: A simple class in Java for generating printable strings containing 128 bits of entropy.

```
import os

def entropic_token(token_size=16, random=os.urandom):
    return random(token_size).encode("hex")
```

Figure 4: A simple function in Python for generating printable strings containing 128 bits of entropy.

3.2 Cookie Discovery Attacks

If the cookie is not well protected, attackers can discover the cookie value. Network-based attackers can use passive network eavesdropping ("traffic sniffing") to read cookie values (and, of course, entire requests and responses) when the application does not use HTTPS. In the exploitation scenarios below, we'll see examples of how attackers abuse weak cookie protections, rendering the use of HTTPS ineffective.

Cross-site scripting (XSS) attacks often focus on stealing the session cookie,[11] for example sending *document.cookie* to the attacker in the *src* of an *img* or *script* tag. When an XSS vulnerability is present, the attacker can insert HTML and JavaScript of their choice into the page, and control the victim's session in any way they want.

Attackers can also discover the values of insecure cookies by tricking the browser into handing them the cookie, such as by DNS poisoning, setting up a malicious server in the domain, or active network attacks (rewriting requests and responses as they traverse the network, owning the router, setting up an impostor router, and so on). In general, DNS and Internet routing services are not guaranteed to be secure, and application developers must not assume that they are. And again, if the application does not use HTTPS, an attacker on the same network as the victim can recover the full text of the requests and responses with a traffic sniffer such as Wireshark[12].

[11] Although that's not all an attacker can use XSS for. Attackers exploiting XSS can rewrite pages to include a fake login form for phishing, conduct CSRF attacks, provide false information, or essentially anything else.

[12] Wireshark is freely available at http://www.wireshark.org/.

3.3 Cookie Setting Attacks

Some types of attackers, such as active network attackers, can set a user's session cookies to a value the attacker controls. This can be effective if the application's session management is flawed. The vulnerability class is called *session fixation*, and the process of exploiting such vulnerabilities is sometimes called *browser priming*.

If the application re-uses a given session cookie when the session transitions from anonymous → authenticated, the attacker can employ various means to set a cookie value of their choice in the client, and then wait for the user to authenticate. Now the attacker knows the value of a session ID for an authenticated session!

To exploit a session fixation vulnerability, attackers can set cookies over HTTP, even for sites that are normally served over HTTPS. If the site grants the user a session cookie and then marks that session as secure, rather than giving users who authenticate a fresh secure session ID, attackers can impersonate the site and give users a session ID known by the attacker prior to authentication. When the user authenticates (even though this is done over SSL and not visible to the attacker) the attacker's known session ID becomes authorized.

Similarly, the attacker can "prime" the browser on a shared machine (such as at an office or on a public kiosk) to manually insert a known cookie value for a vulnerable site.

The same problem can exist if the application re-uses the cookie when the session transitions from authenticated with low-privilege → authenticated with higher-privilege.

"Session Fixation Vulnerability in Web-based Applications" by Mitja Kolšek [13] discusses exploitation scenarios for session fixation. (Kolšek also discusses the guessing, discovery, and setting models of session attack.)

4 Exploitation Scenarios

4.1 A Non-*Secure* Cookie

Most developers are already aware of the fact that on Ethernet (and most other) networks, passive network eavesdropping is trivial. This is of course a major motivating factor for using HTTPS: the encryption foils a passive network observer.

However, if an application has a sensitive cookie for which the *Secure* attribute is not set, the browser will send the cookie in plaintext requests to the server, thus revealing the cookie to anyone listening on the network. While some applications already have a mix of secure and insecure pages in the same application, and some even have mixed secure/insecure content in the same page,[14] an attacker can often force or entice a victim's browser to follow a link to a plaintext resource in the scope of the cookie.

Example: ExampleCo's application, https://app.example.com/, is deployed on an HTTPS server. HTTP requests are redirected to the HTTPS login page. The *Secure* flag is not set on the session cookie. The *Domain* is unset.

Where is the risk? A passive attacker can entice the user, and an active attacker can force the browser, to make a request to the HTTP site. For example, Mallory might have control of a site (such as a blog or message board) that the users of app.example.com are likely to visit[15] that contains HTML like the following:

[13] Available at http://www.acros.si/papers/session_fixation.pdf.
[14] Both of these problems are exploitable vulnerabilities.
[15] Say, a popular blog, or an internal ExampleCo message board or wiki.

```
<img src="http://app.example.com/whatever.jpg" width="0" height="0" />
```

Mallory doesn't even have to be the legitimate administrator of the blog or message board. If the blog has an XSS vulnerability or allows users to use HTML in their postings, she can use that capability to insert the malicious *img* tag.

When Alice visits Mallory's page, the browser does what it was told: it sends the cookie to app.example.com, in the clear. Mallory, who is ready with her network traffic sniffer, swipes Alice's cookie.

If Mallory can't entice users to visit an HTML page under her control, she can also *rewrite* the content of other sites the victim visits to include HTML like the above. She can do this by owning the network infrastructure, for example.[16] After all, *data integrity*, even on a hostile network, is one of the security properties TLS provides — but with plaintext HTTP, there is no guarantee that messages have not been tampered with in transit.

4.2 A *Secure* Cookie

ExampleCo's non-profit tax shelter runs an application at https://app.example.org/. The server only listens for HTTPS on port 443 — no other services are running. The *Secure* flag is set on the session cookie. The *Domain* is ".example.org".

Where is the risk? A passive attacker entices the user, and an active attacker forces the browser, to make a request to another HTTPS server in the cookie's *Domain*.

```
<img src="https://evil.example.org/whatever.jpg" width="0" height="0" />
```

The browser does what it was told: it sends the cookie to any host in the domain.

Mallory sets up her own server at evil.example.org and uses that hostname in the URL. Then she simply accepts delivery of the cookie!

Attackers may be able to get valid, signed certificates for hosts in the domain. One department of the organization may be hostile to another, or another application on another host in the domain (say, blog.example.com) may have an entirely different threat model than the application at app.example.com. For example, a cross-site scripting vulnerability on blog.example.com could spread to attack app.example.com unless the latter keeps its cookies tightly scoped.

On some networks, especially on corporate domains with their own certificate authority, every machine receives a valid hostname — sometimes even a valid SSL certificate — when it connects. These certificates are likely to be trusted by the same machines that are likely to be targets of the attacker and privileged site users, namely the users working at the corporation.

4.3 Not Even Listening on Port 80 but Still Vulnerable

ExampleCo's Tonga branch office runs an application at https://app.example.to/. The server only listens for HTTPS on port 443 — no other services are running. The *Secure* flag is not set on the session cookie. The *Domain* is unset.

Where is the risk? The browser really wants to send the cookie over a plaintext connection — the attacker just has to ask it nicely. (Because the developers and/or deployers did not specify that the cookie should be *Secure*, the browser has no reason to think that it should be.) Since the server is not listening on port 80, the client cannot create a TCP connection to it, and thus cannot send the cookie.

However, note that attackers can always impersonate any insecure port on any server (remember: the network layer provides no security guarantees!). In this case, the attacker does not even need to impersonate a port; instead, the attacker creates a plaintext

[16] At the annual hacker conference DefCon, pranksters like to run rogue wireless access points that are programmed to change every page its users visit to include an offensive picture. Advertisers sometimes use the same technique to insert advertisements into web pages, to help pay for free wifi in some locations I have seen. See Appendix A for an example.

link to port 443 in some page the victim is likely to visit:

```
<img src="http://app.example.to:443/whatever.jpg" width="0" height="0" />
```

After all, the user is likely to be viewing another site at the same time as they are using the sensitive application at app.example.to — for example, using a search engine (over an insecure and hence tamperable connection).

The browser will get an error from the TLS server at app.example.to:443 (an HTTP request is not a valid TLS client hello message), but by then it's too late: the cookie has been exposed in the plaintext request!

Note that to verify this problem, you will need to use Wireshark. Because WebScarab never gets a proper HTTP response to such a malformed request, it doesn't show up in the Summary tab. (You can still see the error message in the Messages tab, however.) See Appendix B for a screenshot of this problem in action.

4.4 Summary of the *Secure* and *Domain* Attributes

This table summarizes how the browser will treat cookies given their *Domain* and *Secure* attributes.

	Secure: Unset	*Secure*: Set
Domain: Leading dot	Cookies sent to any host in the domain, or matching subdomain, over any type of connection	Cookies sent to any host in the domain, or matching subdomain, over a secure connection
Domain: Exact hostname only	Cookies sent to original server over any type of connection	Cookies sent to original server over a secure connection

5 Conclusions and Recommendations

Assume the attacker completely controls the network. Obviously, the wireless network in a coffee shop is not trustworthy, but neither is your user's LAN. The automatic setup and support protocols for the Internet (DNS, Ethernet, DHCP, ARP, and so on) are not designed to be secure, and are all trivially spoofable.[17] On the other hand, TLS is designed to be secure, even in the presence of an adversary who is actively attacking the support protocols. However, you only get the advantages of TLS if you design and implement your application in a way that does not leave the attacker an easier means of exploitation. For session management, this means we must take extra precautions to protect the session identifier: the cookie. The session cookie in a web application is *equivalent to a password* (at least for the duration of its validity). It's the key to a user's account, and therefore must be as strongly protected as a password.

Without secure session management the application, its users, and the sensitive data it manages are extremely vulnerable, regardless of any other protections in place. Most developers and systems administrators understand that SSH provides security guarantees that plaintext protocols like Telnet and rlogin cannot, but web applications with session management problems like those described above are no more secure than Telnet — *even if they are using TLS.*

Attackers can very easily determine if a web application has weak session management. Fortunately, so can you, and with some attention to detail these issues can be resolved.

[17] For example, see Cain and Abel (http://www.oxid.it/cain.html) and Ettercap (http://ettercap.sourceforge.net/).

A Middle-person Attacks Are a Reliable Business Model

Figure 5 shows what Google looks like at a cafe in Austin, TX when using the cafe's open wifi access point. (The internet connection was advertised as being paid for by the ads.) Note that Google does not put ads on its front page. While (presumably) only advertising was inserted into third-party pages, the wifi AP's owner could just as easily have inserted arbitrary content, including XSS attacks, cookie-leaking references (such as image tags, as seen in the Exploitation Scenarios section), or anything else. This advertising is technically indistinguishable from a fatal middle-person attack. (In particular, there can be no assurance that the "Download Google Toolbar" button will download the *real* Google Toolbar...)

Unless the AP's owner also had the private key matching one of the public certificates in the operating system's trusted certificate store, and unless malware has not already been installed on the client, a secure web application would not be affected by any of the threats posed by this type of attack.

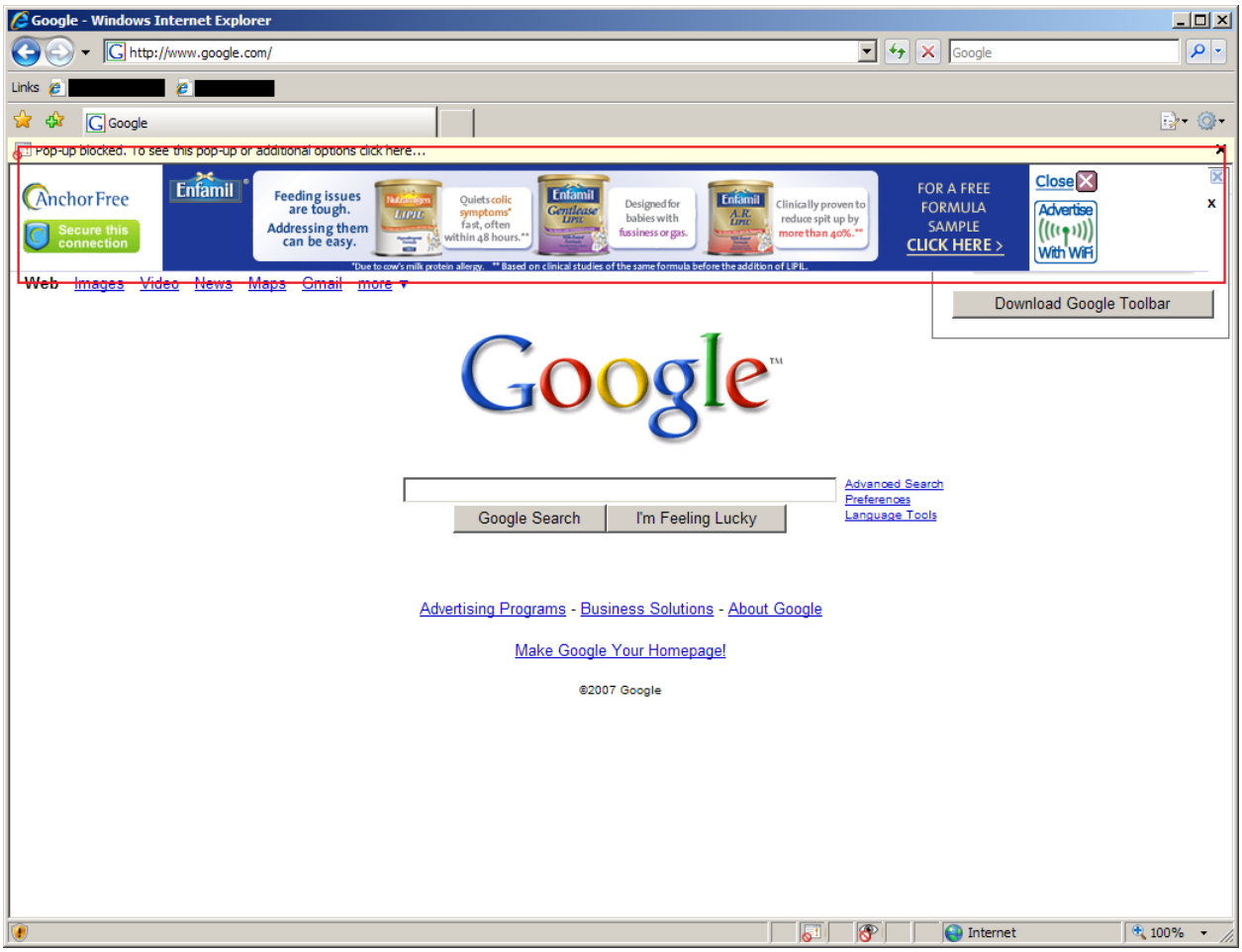

Figure 5: Middle-person attacks, including those performed by subverting the network infrastructure, are reliable enough to be a workable business model.

B Sending Plaintext HTTP Requests to Port 443

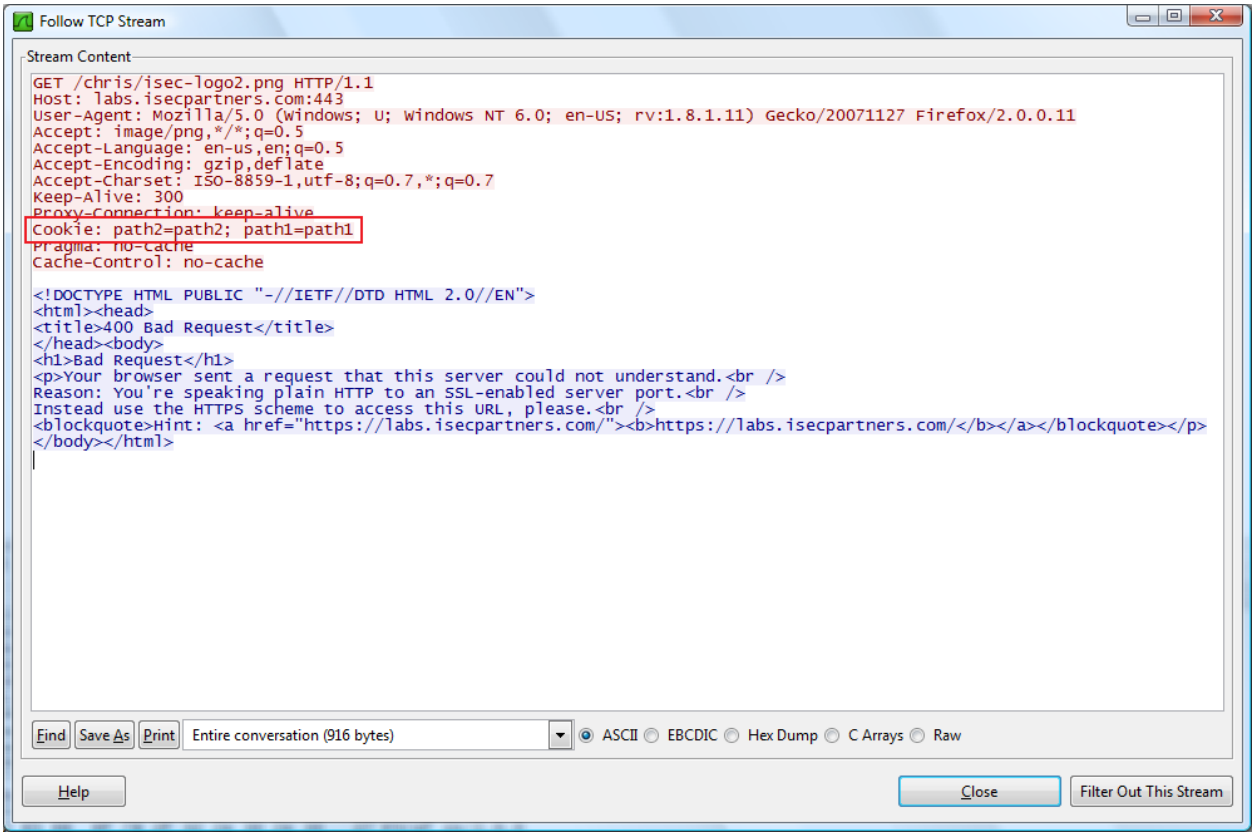

Figure 6: This screenshot shows Wireshark's view of a plaintext HTTP request made to port 443 of a web server.

C Alternatives to Cookies

Some web application frameworks can be configured to communicate the session identifier and/or session state not in a cookie, but elsewhere in requests and responses such as in query strings or POST bodies. This is sometimes called "URL rewriting": normally, the session identifier for these frameworks (for example, a large random number) is stored in a cookie (say, *JSESSIONID*), but when URL rewriting is enabled, a key = value pair like "jsessionid=ABCDE12345" is added to the query string for each request. The application server rewrites the URLs in response documents to contain the *jsessionid* parameter so that the session is maintained.

For example, a request like this:

```
GET http://www.example.com/myapp/index.html HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=3E880015CF879C5014FEAB04C6623203
```

would be reformulated like this:

```
GET http://www.example.com/myapp/index.html?jsessionid=
    3E880015CF879C5014FEAB04C6623203 HTTP/1.1
Host: www.example.com
```

This enables even browsers that do not support cookies (possibly due to user preference) to maintain a session with the application.

However, there is a security problem with this technique: URLs have a marked tendency to leak. For example, users may unwittingly paste URLs into emails or chat, URLs are stored in server logs, URLs are stored in the browser's history, and browsers will expose URLs across sites in the HTTP Referer header. Some browsers include the URL in the footer of a printed page. When this happens, a user's session has been compromised, since the session identifier is right there in the URL. In fact, this is why you should not put any sensitive information in URLs — the problem is more general than session management.

D Securely Implementing Client-side Sessions

To implement secure client-side session storage, we must meet several requirements:

- The total size of the serialized session object must be ≤ 4096 characters, per section 5.3 of RFC 2965.
- The cookie's confidentiality must be protected, so that any sensitive information in the session state is not revealed to attackers (including malicious users).
- The cookie's integrity must be protected, so that forged or mangled sessions are not accepted. Smart attackers can sometimes tweak even encrypted data to change its post-decryption meaning in a malicious way, such as by changing a user ID or privilege level encoded in the cookie.
- The serialization format must be extensible, and backward- and forward-compatible. If the application is upgraded while in production, current sessions must not be lost or invalidated.
- Although we must accept some risk of cookie replay, it must be possible to limit the window of time in which this is possible.
- It should be easy and fast to determine if the serialized session is valid.

With careful use of cryptographic operations we can achieve most of these requirements. (We make the extensibility and compatibility of the serialization format orthogonal to the security mechanism.) The use of a good block cipher, like AES in cipher block chaining (CBC) mode, provides us with strong confidentiality protection (as long as we use a highly entropic encryption key and a unique intialization vector).

Note: It is crucial to use a cipher mode that ensures that each encryption, even when the plaintext and the key are the same, is randomized. CBC provides this by using a unique and random intitialization vector (IV), whereas electronic codebook (ECB) mode does not, resulting in the same ciphertext each time. I have often seen ECB, or CBC with a static IV, used by real (even high-profile) web applications. Non-randomized encryption may leak information about the plaintext that could be useful to an attacker. For more information, see *Practical Cryptography* by Niels Ferguson and Bruce Schneier, pp. 69 – 70:

> Do not ever use ECB for anything. It has serious weaknesses, and is only included here so that we can warn you away from it.
>
> What is the trouble with ECB? If two plaintext blocks are the same, then the corresponding ciphertext blocks will be identical, and that is visible to the attacker. Depending on the structure of the message, this can leak quite a lot of information to the attacker.

We get strong integrity protection by using a message authentication code (MAC) such as HMAC-SHA1,[18] again with an entropic secret key. It is computationally infeasible for an attacker to forge or mangle a cookie in a way that we cannot detect.[19]

By including a timestamp in the cookie, we can implement a session timeout policy by refusing to accept cookies that are older than a certain maximum age. Crucially, the timestamp must also be integrity protected with the MAC so that it cannot be forged by an attacker.

The secret keys for encryption/decryption and HMAC signing must be created using a good source of entropy, such as one of those listed in Section 3.1. They should not be statically embedded into your application's code, but the application should receive them at runtime as configuration parameters. Ideally they are generated automatcally and a human operator never needs to know them.

Figure 7 sketches our scheme for serializing and protecting the session state. (Note that the "||" operator indicates string concatenation.)

Of course, the binary blobs produced by the *HMAC* and *AES_CBC_encrypt* functions will have to be encoded, such as with base-64 encoding, to be transported in HTTP headers. The arbitrary data that is the actual cookie data could be any extensible data format appropriate for your application framework, such as JSON, comma-separated values, URL-encoded key = value pairs, etc.

[18] Keyed-hashing for message authentication (HMAC) is described in RFC 2104: http://www.faqs.org/rfcs/rfc2104.html.

[19] In "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm" Bellare and Namprempre describe the theoretical basis for this type of confidentiality and integrity mechanism.

cookie := *hmac_signature* || *timestamp* || *data_blob*
timestamp := milliseconds since epoch
hmac_signature := *HMAC*(*secret_key*, *timestamp* || *data_blob*)
data_blob := *AES_CBC_encrypt*(*secret_key*, *random_IV*, *compress*(arbitrary data))
secret_key, *random_IV* := output from a cryptographic random number generator
session_timeout := duration of a session's validity in milliseconds

Figure 7: Sketch of a scheme that provides confidentiality, integrity, and expiration for arbitrary data, including serialized session states.

The application validates a session cookie as follows:

1. Decode *hmac_signature*.
2. Compute *HMAC*(*secret_key*, *timestamp* || *data_blob*) and compare it to *hmac_signature*. Fail if they differ.
3. Decode *timestamp*.
4. Verify that the current time in milliseconds since the epoch is not greater than *timestamp* + *session_timeout*.

If the cookie is not valid, the application must refuse the requested action and redirect the user to the login page.

If the cookie is valid, the application can

1. decrypt *data_blob*;
2. decompress *data_blob*; and
3. parse or deserialize *data_blob* as appropriate.

At this point, the application has a valid state object for the user's session and can proceed with processing the requested action.

E Securely Implementing Client-side Sessions

This appendix lists the code for the cookie attribute tests.

File: cookie-attributes.php

```
<?php
$expiry = time() + 100000;

setcookie("default", "default");
setcookie("expires1", "expires1", $expiry);
setcookie("expires2", "expires2", time() + 100);
setcookie("path1", "/", $expiry, "/");
setcookie("path2", "/chris", $expiry, "/chris");
setcookie("path3", "/chris/cookie-test", $expiry, "/chris/cookie-test");
setcookie("path4", "/gargle-bargle", $expiry, "/gargle-bargle");
setcookie("domain1", "isecpartners.com", $expiry, "/chris/cookie-test",
          "isecpartners.com");
setcookie("domain2", ".isecpartners.com", $expiry, "/chris/cookie-test",
          ".isecpartners.com");
setcookie("domain3", "labs.isecpartners.com", $expiry, "/chris/cookie-test",
          "labs.isecpartners.com");
setcookie("domain4", "toes.isecpartners.com", $expiry, "/chris/cookie-test",
          "toes.isecpartners.com");
setcookie("domain5", ".labs.isecpartners.com", $expiry,
          "/chris/cookie-test", ".labs.isecpartners.com");
setcookie("secure", "secure", $expiry, "/chris/cookie-test",
          "labs.isecpartners.com", TRUE);
?>

<h1>Cookie Attribute Test</h1>

<p><var>document</var><tt>.</tt><var>cookie</var>: <span
id="read-cookie"></span></p>

<p>In the above, you should see only those cookies which are in the scope
(<em>Path</em> and <em>Domain</em>) of this page.</p>

<script>
document.getElementById("read-cookie").innerHTML += document.cookie;
</script>

<p>Click each link below. In WebScarab, watch the browser make the requests,
and note which cookies are sent in each request. You may also want to use
the Add &lsquo;n&rsquo; Edit Cookies Firefox extension to see which cookies
your browser accepts (it may not accept them all, such as when
labs.isecpartners.com tries to set a cookie with <var>Domain</var> =
&ldquo;toes.isecpartners.com&rdquo;). Expect the behavior to vary between
browsers and between different versions of the same browser.</p>

<p>Note that there are no DNS entries for zip.labs.isecpartners.com,
evil-impostor.isecpartners.com, and toes.isecpartners.com, so you'll have to
add an entry in your hosts file.  (On Linux/Unix the hosts file is
/etc/hosts, while on Windows it is %SystemRoot%\system32\drivers\etc\hosts.
%SystemRoot% is usually c:\windows.) You can set it to anything, such as to
```

```
your own web server. You will get a 404, which is ok for the purposes of
this test.</p>

<p>Here is the line to add to your hosts file (it should all be on one line):</p>

<pre>72.52.84.219 zip.labs.isecpartners.com toes.isecpartners.com
    evil-impostor.isecpartners.com</pre>

<ul>

<li><a
href="http://labs.isecpartners.com/chris/cookie-test/isec-logo.png">
http://labs.isecpartners.com/chris/cookie-test/isec-logo.png</a>

<li><a
href="https://labs.isecpartners.com/chris/cookie-test/isec-logo.png">
https://labs.isecpartners.com/chris/cookie-test/isec-logo.png</a>

<li><a
href="http://toes.isecpartners.com/chris/cookie-test/isec-logo.png">
http://toes.isecpartners.com/chris/cookie-test/isec-logo.png</a>

<li><a
href="https://labs.isecpartners.com/chris/isec-logo.png">
https://labs.isecpartners.com/chris/isec-logo.png</a>

<li><a
href="http://zip.labs.isecpartners.com/chris/isec-logo.png">
http://zip.labs.isecpartners.com/chris/isec-logo.png</a>

<li><a
href="https://evil-impostor.isecpartners.com/chris/isec-logo.png">
https://evil-impostor.isecpartners.com/chris/isec-logo.png</a>

<li><a
href="http://morecowbell.cybervillains.com/chris/isec-logo.png">
http://morecowbell.cybervillains.com/chris/isec-logo.png</a>

<li><a
href="http://labs.isecpartners.com:443/chris/isec-logo2.png">
http://labs.isecpartners.com:443/chris/isec-logo2.png</a>

</ul>

<p>You might also be interested in <a href="/chris/httponly/">a test of the
<em>HttpOnly</em> flag</a>.</p>
```

File: httponly-test.php

```
<?php

header("Set-Cookie: Noodle=doodle; HttpOnly; Path=/chris/httponly");

?>

<h1>Testing the <em>HttpOnly</em> Attribute</h1>
```

```
<p>The cookie: <span id="read-cookie"></span></p>

<p>The cookie after being written to: <span id="write-cookie"></span></p>

<script>

document.getElementById("read-cookie").innerHTML += document.cookie;
document.cookie = "Overwritten!!";
document.getElementById("write-cookie").innerHTML += document.cookie;

</script>

<p>In Firefox 2.0.0.11, for example, you&rsquo;ll see that JavaScript can
prepend a value to <var>document</var><tt>.</tt><var>cookie</var>, and can
read what it prepended, but cannot read or overwrite the server-provided
value. When you reload the page, you will see that the browser sends both
the original value and the new value.</p>
```