Half-Baked Cookies: Client Authentication on the Modern Web

Yogesh Mundada[1], Nick Feamster[1], Balachander Krishnamurthy[2], Saikat Guha[3], and Dave Levin[4]

[1]School of Computer Science, Georgia Tech
[2]AT&T Labs – Research
[3]MSR India
[4]University of Maryland

Abstract

Modern websites set multiple authentication cookies during the login process to allow users to remain authenticated over the duration of a web session. Web applications use cookie-based authentication to provide different levels of access and authorization; the complexity of websites' code and various combinations of authentication cookies that allow such access introduce potentially serious vulnerabilities. For example, an on-path attacker can trick a victim's browser into revealing insecure authentication cookies for any site, even if the site itself is always accessed over HTTPS. Analyzing the susceptibility of websites to such attacks first requires a way to identify a website's authentication cookies. We developed an algorithm to determine the set of cookies that serve as authentication cookies for a particular site. Using this algorithm, which we implemented as a Chrome extension, we tested 45 websites and found that an attacker can gain access to a user's sensitive information on sites such as GoDaddy, Yahoo Search, Comcast, LiveJournal, stumbleupon, and Netflix. In cases where these sites cannot enable site-wide HTTPS, we offer recommendations for using authentication cookies that reduce the likelihood of attack. Based on these recommendations, we develop a tool, Newton, that website administrators can use to audit the security of a site's cookie-based authentication and users can run to identify vulnerabilities at runtime.

1 Introduction

The web's core protocol, the Hypertext Transfer Protocol (HTTP), is inherently stateless; to manage higher-level application state, web applications commonly store information about user sessions in "cookies"; each website sets a collection of cookies on a user's local machine, which the user's browser sends to the server hosting the website on each subsequent request. Cookie contents are diverse and may contain information ranging from user preferences (*e.g.*, does the user prefer a white or black background) to locality (*e.g.*, what language the site should be rendered in) to transaction state (*e.g.*, what is currently contained in a user's shopping cart). Additionally, many websites use cookies to *authenticate* users to a website, and various cookies may also indicate to the server whether or not a user is currently logged into the website.

Customarily, a user may initially log into a website using a username and password. To prevent the user from having to re-enter these credentials in subsequent interaction with the website, the server typically sets *authentication cookies (auth-cookies)*, which the browser automatically returns with each subsequent HTTP request until the user's session expires. The web server hosting the sit checks these auth-cookies to determine whether any particular HTTP request is authorized to perform the operation associated with that request (*e.g.*, access a particular webpage, add an item to a shopping card, obtain personal profile details). Thus, once a user initially authenticates to a web server, the auth-cookies are a critical security linchpin: In many cases, access to these auth-cookies gives an attacker complete control over a user's account.

Individual cookies contain attributes such as `secure` (which prevents snooping cookies over insecure connections) and `HTTPonly` (which prevents cookie theft via cross-site scripting), but, the complexity of modern websites and services makes it difficult to secure all of these cookies. Popular websites sometimes have incorrect security settings on auth-cookies; for example, in May 2014, the incorrect security settings of WordPress's auth-cookies left users vulnerable to session hijacking attacks [2]. Although the public is becoming increasingly aware of these types of vulnerabilities, our results show that they remain prevalent in practice.

In the past, web applications were simple and would use only a few auth-cookies (in most cases, only one) to control access to the account. Researchers already have published recommendations to make web sessions secure in these cases [13]. Yet, modern web applications are significantly more complex. These applications often have millions of lines of code and reuse legacy code from other components; they are designed and implemented by large teams of developers are frequently modified. Moreover, today's web sites and applications are multi-faceted, and "login" or "authenti-

cation" is no longer strictly binary: For example, a user may have different authorization or access to different parts of a web site (*e.g.*, a user might have the ability to view account balances but not to execute trades, or add items to a shopping cart but not purchase items or ship to a different address).

The complex nature of authorization for modern web applications makes it increasingly common for these applications to use *multiple* authentication cookies, the combination of which determine the user's login status or ability to access different parts of a web site. More than half of the 45 sites that we analyzed set more than 20 cookies, and more than ten of these sites use multiple authentication cookies. Websites use multiple authentication cookies for several reasons: Some portions of a website may be loaded over HTTPS, while other portions of a site may be loaded over cleartext HTTP; a user on the same site may operate under different roles (*e.g.*, administrator, user, site owner, moderator); a site may have many different services or "properties" under the same domain; and finally, the cookies that a site uses to authenticate a user may change regularly. For example, Google has multiple applications including mail, search, and calendar; we find that each of these offerings uses different combinations of authentication cookies. Sometimes, the combinations change regularly: for example, the authentication cookies for Gmail have already changed three times since December 2013.

The evolving nature of authentication cookies in modern web applications begs for a re-appraisal of the use of auth-cookies for web authentication and the potential vulnerabilities that their misuse may induce on real sites. In light of modern web applications and websites, we must also develop new recommendations for best practices when using auth-cookies to authenticate users on modern websites. The misuse of authentication cookies in a web application's design can result in unintended security vulnerabilities, particularly in certain settings where a user may also be vulnerable to traffic snooping or cross-site scripting vulnerabilities. Because different parts of the application can be accessed over HTTP (non-confidential content), or HTTPS (confidential content), a web application developer may not set the `secure` attribute on some authentication cookies if they would be sent on normal HTTP connection. Each such decision of whether to set the `secure` flag on an auth-cookie may be correct *individually*, but due to complex access control code paths, an attacker may be able to gain access to privileged information with access to only a subset of cookies whose `secure` attribute is not set. In scenarios where an intermediary is on-path (*e.g.*, any open WiFi network, such as a coffee shop, airport, or even a friend's house), an attacker might gain access to all cookies whose `secure` attribute is not set. In these settings, for many web sites, we demonstrate that an on-path entity can gain all of the information needed to compromise a user's account for a given web application.

Before deploying a web application, a web administrator should assess and eradicate the vulnerabilities that result from cookie-based web authentication. Similarly, users can receive warnings when a web session is vulnerable to auth-cookie hijacking. Both cases require identifying the auth-cookie combination that authorizes a user to perform a specific operation on a site and auditing those auth-cookies for vulnerabilities. Although it might appear simple for an application developer or website administrator to simply check the source code of an application for such vulnerabilities, tracing all possible execution paths through complex web applications that set many cookies is a challenging problem in and of itself. We treat the web application as a black box, which makes our approach applicable for both website administrators and users.

The naïve approach to determining the specific combination of cookies that determines a user's authorization to perform some action is combinatorial. We develop an algorithm to reduce the time for computing the authentication cookies for a site to polynomial time and incorporate this algorithm into a tool, *Newton*, which helps website administrators audit their sites for potential vulnerabilities.

This paper presents the following contributions:

- We develop an algorithm to discover the auth-cookie combinations that will authenticate a user to any service on a website. Our algorithm also derives the unique auth-cookie combinations that allow a user to access different sub-services (or "properties") on the same website. We show how using this algorithm helps us identify the auth-cookies combinations in practical time.
- We implemented the algorithm to discover auth-cookies for a site as a Chrome browser extension, Newton. The tool treats each web application as a black box and operates without any prior or privileged knowledge about either the website or the user. Administrators can use Newton to run security audits on installed Web applications (whether or not they have source-code access), and users can avoid or curtail interaction with sites that are vulnerable to session hijacking.
- Using Newton, we analyzed combinations of auth-cookies to audit 45 websites; we found 29 sites where the use of auth-cookies is insecure in some way. Many of these vulnerabilities arise because of logic errors in how the application handles auth-cookies. For example, on 18 different websites, the web application does not invalidate old auth-cookies when the user logs out, leaving the user vulnerable to replay attacks.
- We analyze the vulnerabilities we discovered to develop specific recommendations for designing more secure client authentication mechanisms using auth-cookies.

The rest of the paper is organized as follows. Section 2 provides background on Web cookies and cookie-based authentication. Section 3 discusses our specific goals to make our solution scalable for large number of users and sites and the challenges we faced in doing so. Section 4 details our solution and our insight to reduce the effort in verifying the search space. In Section 5, we describe implementation challenges to make the solution deployable across many users. Section 6 presents the auth-cookie combinations that we automatically

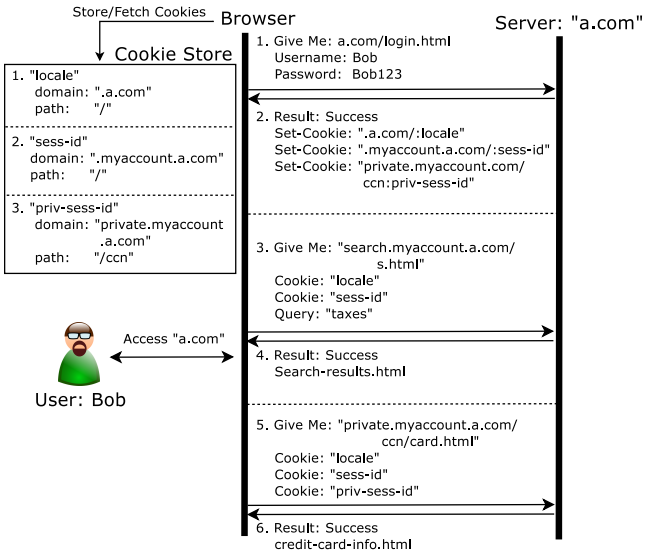


Figure 1: *The general operation of Web cookies.*

discovered for major services. Section 7 discusses both the limitations of our approach and avenues for future work. Section 8 discusses related work, and Section 9 concludes.

2 Background

We present general background on web cookie operation and how different cookie attribute settings can affect the security of web client authentication. We then describe authentication cookies and scenarios where these cookies might be stolen or otherwise compromised. Finally, we present an overview of an algorithm to determine which cookies on a website correspond to the user's authentication cookies.

2.1 General Web Cookie Operation

An HTTP cookie is the state management mechanism that HTTP has used since its initial design. A cookie has properties such as name, value, flags, expiration data, and a match rule. A match rule has a domain and path components. The web server when sets cookies and the respective values for each cookie in HTTP responses using the `Set-Cookie` directive. Whenever a browser makes an HTTP request to a URL, the browser first compares that URL against all present cookies' match rules for that domain. The browser will then automatically send those cookies whose match rules evaluate to true for that URL. To determine whether a cookie's match rule is satisfied, the browser first compares the domain name in the match rule with the domain in the URL of the HTTP request. If the domain of the cookie starts with ".", then any subdomain of the subsequent domain will match; otherwise, the cookie only matches if the domain name of the URL exactly matches the domain in the cookie. If the domain matches, then the browser evaluates whether the path specified in the cookie matches the path in the URL; if a match exists for both the domain and path attributes, then the browser returns the cookie with the HTTP request.

Figure 1 shows the process where a user Bob requests a service on `a.com` to login andza access a resource. After Bob has logged in, the site sets the `sess-id`, `locale`, and `priv-sess-id` cookies. The `locale` cookie has a match rule with domain and path attributes that match any requested URLs on `a.com`. Thus, the browser will send this cookie with any HTTP request that the browser sends to `a.com`. On the other hand, the `sess-id` cookie has a domain match part equal to `.myaccount.a.com`, and thus is only returned with requests for `myaccount.a.com` or any subdomain (*e.g.*, `search.services.myaccount.a.com`). Since both cookies' path match attributes are `/` the cookies match all paths. The more restricted `priv-sess-id` cookie will match against the exact domain `private.myaccount.com` and paths such as `/ccn` or `/ccn/card1`.

Each cookie also has two boolean attributes, `HttpOnly` and `Secure`, which further restricts a web site's ability to access a particular cookie. A web application's Javascript code, which runs in a user's browser, can read, write, and delete cookies created by the same domain [5]. If the cookie's `HttpOnly` flag is set to true, however, then even Javascript from the same domain cannot read that cookie. The `HttpOnly` mechanism thwarts potential XSS exploits for the site from pretending to have originated from the targeted site, reading its sensitive cookies and leaking them to an attacker. Cookies also have a `Secure` attribute; if this attribute is set, then the browser will only return the cookie over an SSL/TLS channel. After a cookie's expiry date, the browser will automatically purge the cookie from the cookiestore. If the web server does not set an expiry date on a cookie, then the cookies is a *session cookie*, which the browser will automatically deleted when the user quits the web browser application. In practice, most users rarely terminate their web browsers, so session cookies can persist for long periods of time.

2.2 Authentication Cookies ("Auth-Cookies")

A web server typically sets authentication cookies ("auth-cookies") when a user initially authenticates to the web server (*e.g.*, with a username and password). These cookies allow a user to perform various privileged operations on a site; different combinations of auth-cookies may authorize different actions on a website. For example, in Figure 1, `sess-id` is an auth-cookie that allows the user to access profile information, whereas cookie `priv-sess-id` allows access to more sensitive information, such as a credit card number. Auth-cookies also ensure that a user's session remains continuous in case connectivity is interrupted (*e.g.*, due to termination of a TCP session or a change in the user's IP address). Because these cookies allow the user to perform various operations without re-entering credentials, they are extremely valuable to attackers.

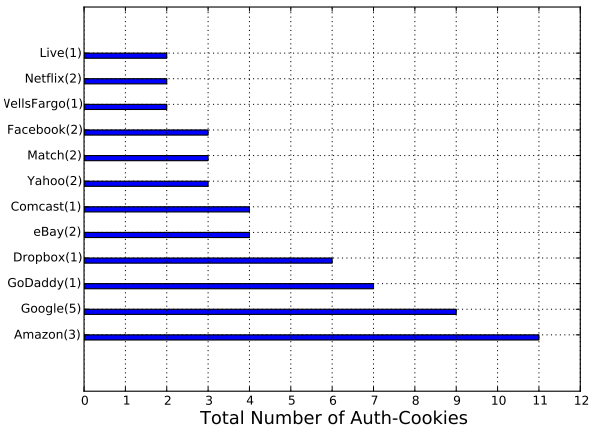


Figure 2: *The cumulative number of auth-cookies present on explored different sub-services on various websites. The number of sub-services that we explored on each site is given in parentheses next to the site name. Many other sites that we have not shown used two auth-cookies.*

2.3 When Auth-Cookies Can Be Stolen

Of course, any on-path attacker can sniff cookies that are not encrypted via HTTPS. Thus, auth-cookies that are sent over unencrypted channels (*e.g.*, cleartext HTTP) are cause for concern. Additionally, a vulnerable DNS server would allow an attacker to poison its cache and reroute victim's traffic to machines under attacker's control [3]. Below, we present several plausible scenarios where an attacker can cause the user's browser to divulge auth-cookies for a specific site.

Man-in-the-middle attack with DNS traffic manipulation. An on-path attacker or any attacker who can control and manipulate DNS traffic destined for a user's machine—as is the case with untrusted WiFi hotspots, DNS servers that are vulnerable to cache poisoning, or rogue ISPs—can induce the user's browser to submit all auth-cookies without the `secure` flag across *any* site where the user is logged in.

To illustrate the subtlety of this attack, suppose that the victim is logged into a Yahoo mail accout and always accesses `mail.yahoo.com` over HTTPS; suppose also that that the `secure` flag is not set on the `sess-id` auth-cookie for `mail.yahoo.com`. In this case, HTTPS provides a false sense of security: To hijack the user's session, the attacker might resolve DNS lookups for the domains of embedded resources (*e.g.*, advertisements) to (or through) a machine that is under the user's control. The attacker can then inject new content in place of existing ads, such as `<img>http://mail.yahoo.com/gotyou.jpg</img>`. Even generally reputable ISPs have performed such attacks in the past: for example, a large French ISP, Orange, was performing this type of ad substitution [18]. Once the victim's browser receives this modified advertisement, his browser will try to fetch this image causing the browser to issue an HTTP request to retrieve the image. At this point, the user's browser would send the `sess-id` with this request in plaintext.

Compromised third-party Javascript. Javascript that a webmaster hosts on a webpage has full permissions and access to cookies for that domain unless the cookies have the `HttpOnly` attribute set. In these cases, if auth-cookies do not have this attribute set, an attacker may be able to read the cookies for that domain. This scenario may arise in cases when a webmaster includes Javascript from an untrusted or third-party developer (*e.g.*, site owners sometimes include Javascript for purposes of tracking, analytics, or functionality); in other cases, the site that hosts a webpage may be compromised, allowing third parties to include Javascript on the host page. Another significant problem is fetching Javascript code from expired or mis-typed domains by web applications not maintained properly. If an attacker registers such a domain and starts hosting malicious Javascript, then all the users of site that include this Javascript become vulnerable [21].

Cross-site scripting. An XSS vulnerability exists whenever one site trusts user input and presents it to another site without sanitizing it first. These attacks are significant when input from one user would be exposed to another user. Prime candidates for these attacks are social networking sites where users interact with each other frequently. Even on financial sites such as banking, one user may send a message to another while transferring money. We acknowledge that mounting this attack is challenging because vulnerabilities are site-specific. Yet, considering that XSS is one of the most widely present web vulnerabilities and has been discovered in almost all top websites [23], it is still worth considering that an attacker may be able to steal cookies via an XSS attack.

2.4 Why Existing Defenses Are Insufficient

It is reasonable to think that perhaps all cookies should have the `secure` (or at least `HttpOnly`) attribute set, to prevent attacks such as the ones we have outlined. Unfortunately, such an approach is too coarse. Setting `HttpOnly` on all cookies is not a viable approach because some Javascript occasionally needs legitimate access to a user cookie. One example are "double submit cookies", a mechanism to prevent cross-site request forgery whereby a web client sends a random value in both a cookie and a request parameter and the server checks that the two values are equal [24]. Additionally, setting `secure` on all cookies is not practical because sometimes (often for performance reasons) an application may serve content over both HTTP and HTTPS.

A security mechanism called HTTP Strict Transport Security (HSTS) defends against certain types of man-in-the middle attacks by forcing certain HTTP transfers to occur over HTTPS, whenever possible. Unforunately, HSTS is not a silver bullet solution. Forcehttps [19], which inspired HSTS, acknowledges that due to various complexities of a website's code, deploying HSTS would require very careful analysis of a site, shifting of subdomains wherever necessary; sometimes,

reorganizing a site to use HSTS without breaking the site's functionality is not even possible. Even if HSTS is deployed, it may not be configured correctly and can thus still be vulnerable to SSL downgrading attacks [15]. As a result of these practicalities, HSTS is still not widely deployed: according to one report from July 2014 [27], only 1,756 sites out of approximately 150,000 popular sites had deployed HSTS. Furthermore, some browsers such as Internet Explorer do not even support HSTS [17].

Because simple coarse grained approaches are not sufficient, users, web application developers, and website operators need better tools to compute a site's auth-cookies to better assess and defend against potential attacks. The first step in protecting a user's auth-cookies involves determining which cookies are auth-cookies in the first place. The next section explains why this problem is challenging.

3 Computing Auth-Cookies

We want to compute auth-cookies both at design time (to help website administrators) and at runtime (to help users); these requirements impose significant challenges. We aim to design a tool for computing auth-cookies that is both *general* (*i.e.*, it should compute auth-cookies on as many sites as possible) and *transparent* (*i.e.*, it should not interfere with a user's active sessions to any site). In this section, we outline the challenges that we faced in realizing this goal.

3.1 Strawman Approach

Given a general webpage and cookies set by the webpage's domain, we want to find out which combination of cookies will allow access to that webpage. To do so requires generating combinations of cookies and fetch webpage under test with only those cookies enabled. At a high level, our solution proceeds as follows:

1. fetch a webpage without suppressing any cookies,
2. generate combination of cookies to be disabled and then fetching a webpage with rest of the cookies enabled.
3. compare the response page with the one fetched in Step 1.
4. if the webpage is not fetched successfully, then we mark the disabled cookies combination as a required combination to access that webpage.

Unfortunately, determining whether the page fetch succeeded is difficult; it is also application-specific. For example, failure modes include showing error message, redirecting the user to a login page, or redirecting to default front page. One could use various statistical or image comparison techniques to compare the two pages, but the threshold to define statistically significant results will still vary from site to site. For example, on a site such as Google Mail, the difference between fetched default webpage at `mail.google.com` when auth-cookies are suppressed vs. when they are not, can be detected using a statistical or image comparison algorithm. On the other hand, on sites like Amazon, there is practically no difference between fetched default webpage `www.amazon.com` whether one suppresses auth-cookies or not, because Amazon is an e-commerce web site that shows a user product advertisements on the front page whether the user is logged in or not.

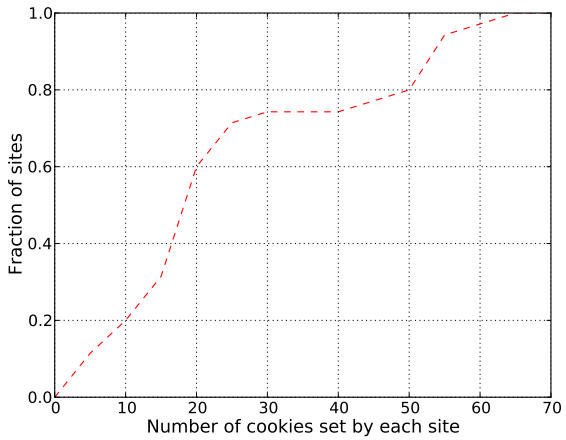


Figure 3: *A distribution showing the number of cookies that each of 45 popular websites set on a user's machine. Half of these sites set more than 20 cookies.*

We observe that on most sites, when a user is logged in and the browser sends all cookies to the server, the username is usually present somewhere at the top of the page or on the left side of the page. Thus, testing whether an HTTP request was performed as a logged-in user reduces to checking whether the user's username is present somewhere on the page where usernames normally appear. Of course, there are exceptions that to not conform to this heuristic, but we found this heuristic to work very well on a large number of sites; our implementation can always be extended to handle these less-frequent corner cases. Naturally, making such an algorithm work in practice entails a considerable set of additional challenges, which we now describe.

3.2 Computing Auth-Cookies is Hard

Many practical complications require us to adapt the basic approach we described. In this section, we describe the practical challenges assoicated with computing auth-cookies.

3.2.1 Many possible cookie combinations

The search space is exponential. A browser often handles at least 50 cookies per domain, with the size of each cookie as large as 4,096 bytes [4]. Figure 3 highlights the complexity of this problem, showing a distribution of the number of cookies that 45 different popular websites set—more than half of the sites set 20 cookies. In the case of some domains such as Google, we saw as many as 130 cookies being set for a single domain. For many sites, any one of multiple combinations of auth-cookies can allow a user to access a site. Determining all possible combinations of auth-cookies that give a user access to a site or service is thus exponential in the number of cookies that the site sets. If a site sets 50 cookies (as in the case of Google, which sometimes sets as many as 120

cookies), then assuming a page load time of 200 ms, then testing all combinations could take about 7 million years.[1]

Focusing only on login cookies doesn't work. One might assume that the cookies that serve as auth-cookie combinations will only be set during login process. This turns out to be false. In other words, it might be reasonable to assume that when a user visits a website, enters a password, and successfully logs in to the website, the cookies set during this process—the "login cookies"—will contain the auth-cookies. For example, when a user logs in to Google, we observe that usually between 14–20 cookies are set, compared to the 80–120 cookies that the site sets over the course of a user's interactions with the site. If we only searched for auth-cookies combinations by exhausting all combinations of the login cookies, we would significantly reduce our search space. Yet, for authorization to Google Calendar, the auth-cookie is only set *after user visits that part of the site* but is set without requiring the user to log in again. (The auth-cookie for Google Calendar is *not* set during the normal login process in Google.) In another case, Amazon does not delete or change many of the auth-cookies after a user logs out. These auth-cookies will not be explicitly set during subsequent logins, so simply searching among the set of cookies that are set at login time does not work.

Identifying auth-cookies by cookie properties doesn't work. One might assume that if a cookie value looks random, is "long enough", and changes across multiple login sessions then it can be assumed to be a part of auth-cookie combination. On the other hand, if the auth-cookie value does not change across login sessions or does not have enough entropy, then it is not a part of auth-cookies. Unfortunately, we observed that some cookies that do not satisfy these properties are still part of the auth-cookie combination. For example, in case of `Facebook`, the `c_user` cookie is a part of the auth-cookie combination; yet, this cookie appears to serve as some type of a user ID and does not change across login sessions. Similarly, in the case of `Twitter`, the `_twitter_sess` cookie is set to some opaque 200-character string that changes across login sessions. Its name and its changing value suggest that it may be an auth-cookie, but it is not. Instead, the `auth_token` cookie whose 40 hex character value remains constant across login sessions is the sole auth-cookie!

3.2.2 Different websites with unique designs

Computing auth-cookies for as many sites as possible requires a field deployment with real users. It is easy to generate auth-cookie combinations in controlled laboratory experiments using a modified browser environment such as Selenium. This approach makes it possible to compute auth-cookie combinations for sites where we *already* have accounts, but does not allow us to compute auth-cookies for the large number of sites where we do not have (or do not want to generate) accounts. For many sites that we aim to study, we might not even be able to obtain accounts because account membership would first require affiliation with some institution (*e.g.*, an account on a banking site would first require having a bank account with that institution). The requirement for a field deployment introduces its own set of challenges, which we describe below.

Determining the URL to use for computing a site's auth-cookies is difficult. On many sites, if the browser visits a website without a specific path, the browswer is redirected to a page where the username is visible, which allows us to complete our test. On other sites, even if the user is logged in, if we test an incorrect URL (or even the correct URL with incorrect parameters), the web server may redirect to a webpage with anonymous context or even log the user out entirely, which not only prevents us from running our test but could also disrupt the user who has installed the tool.

Sending incorrect cookies can disrupt the user's experience. Sending a website incorrect cookies can cause the website to log a user out, disrupting the user session. For example, `Facebook` has an auth-cookies combination '`xs` AND `c_user`', and dropping `xs` resets the `c_user` cookie and logs the user out. Thus, we needed to devise a mechanism to send arbitrary cookie sets from a real user's browser *without disrupting the user if we send an incorrect or invalid set of cookies*. To do so, we run all tests using an in-memory set of shadow cookies that mimic the user's real set of cookies.

Testing by removing cookies from HTTP requests is not effective. Because a browser automatically sends present cookies stored in the user's browser, a tool could perform tests by temporarily suppressing cookies from HTTP requests. Unfortunately, doing so runs the risk of interfering with the rest of the user's normal interactions on the site while the test is underway. Additionally, occasionally a server will detect the absence of a cookie, set it once again in HTTP response, and redirect the browser to originally requested page. This behavior can interfere with testing. Finally, recall that the tool's test for successful page load is the presence of usernames. Yet, a user can have various profile names on different sites that may be different from the username they enter to login, and we cannot expect users to configure all usernames that he has used on various sites.

3.2.3 Timing and transient behavior

Transient failures actions can appear as failed login attempts. The tool might wrongly interpret a failure to successfully load a login page as a result of a network or server failure as a login failure. To mitigate some of these effects, factors such as the client processor and memory load, as well as

[1] This problem is NP-hard. A SAT-problem with 'N' variables can be solved in polynomial time by an oracle for this problem. For this, we can construct a privileged operation that is performed by the web server only if the SAT-equation is satisfied. The browser also has 'N' cookies corresponding to 'N' variables. Assignment to each of the 'N' variables at the server side is decided by which cookies are present in the HTTP request. An oracle can decide the existence of the auth-cookies combinations to execute the privileged operation ultimately solving the SAT problem in polynomial time.

network reachability, should also be taken into consideration before starting off this investigation.

A user may log out during a test. For the tool to compute auth-cookies for a site, a user must be logged into the site in the first place. Yet, in the absence of a successful log in event, detecting whether a user is logged in is difficult. Additionally, a user might log out in the middle of the investigation, or a session might timeout. In such cases, if the tool notices that a user is logged out but does not determine that the logout was due to a user-initiated action, the tool might wrongly infer that certain cookies were part of the set of auth-cookies. In the case of a session timeout, information from in-progress tests might also be lost.

3.2.4 Usability Concerns

Because we want our algorithms to be useful to both website administrators and users, we need to consider various usability concerns, such as when to execute a particular test and what action to take if our algorithms detect a vulnerability. One of the most important concerns is how to compute auth-cookies at runtime without intefering with a user's normal browsing session. We perform a backup of the user's cookies and run tests using a "shadow" cookie store to ensure that testing never interferes with a user's actual session. The tool also only performs testing against a particular website when it determines that a user is not accessing that site. We also provide the user with various options to control both the frequency of tests and the types of mitigating actions to take if the tool detects a vulnerability.

4 Newton: Computing Auth-Cookies

In this section, we describe the design of Newton, a tool that we have built to help website administrators in auditing and common users to protect their web sessions. We refine the basic idea that we presented in Section 3.1 to address the various practical concerns that we raised in Section 3.2.

4.1 Detecting Login Status: Username Presence

Newton uses the presence of the user's name on the webpage to determine that the user is logged in. To detect the presence of a user's name without explicitly asking the user (which is error-prone in any case, since the webpage may display a real name to a logged-in user, rather than the username), we developed a *domain specific language* that allows us to specify how to scrape a user's username and other information from different sites. Using this language, we developed scraping scripts for many popular top sites, including `Facebook`, `Google+`, `Amazon`, and many dating sites to determine a user's real name and username.

Determining a user's login status on a broad range of sites using the presence of a username or real name on the page requires multiple bootstrapping steps. First, we create dummy accounts on an initial "seed" set of sites to help us codify the relationship between the site's structure and the location on the site where the user's name resides. Knowledge of these relationships allows us to develop scripts in our domain specific language to scrape real names and usernames from these sites for any user who installs Newton. We then ran Newton for each webpage for each of the sites where we can determine these relationships. Newton then determines the auth-cookies for each of these webpages (using the algorithms that we describe below). After the user installs Newton, the tool can determine whether a user is logged in on these sites by examining whether the appropriate auth-cookies are set; it can then determine the user's name (or username, depending on the site) by scraping the page. Finally, with knowledge of the user's name(s) and username(s) from these seed sites, Newton can then determine whether a user is likely logged in on other sites (*e.g.*, banking sites) simply by checking for the user's name (or username).

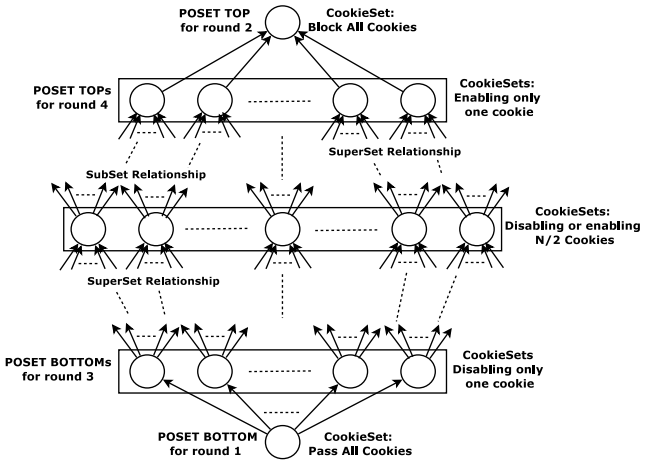


Figure 4: *Bounded complete partially ordered set (POSET) representing the set of all cookie-set combinations that Newton would have to test without optimizations. In each testing round, Newton alternately tests current "tops" or "bottoms" of the POSET. After each testing round, tested cookie-sets are removed from the POSET.*

Using our domain specific language, we have created 70 scraping scripts for top sites where users have accounts usually such as social networking, mailing, utility sites. Using these, we are able to comprehensively cover different variations in a specific user's usernames and thus detect these usernames on sites, even on sites for which we have not written a scraper. However, even if we fail to detect some user's username on some site, Newton can *still* determine the auth-cookie combination for that site by performing the auth-cookie computation for a user who we are able to identify for that site. Because the auth-cookies that a site uses are the same for every user, Newton can build a knowledge base of auth-cookies for a large set of sites that is then shared across Newton users.

4.2 Tackling Combinatorial Explosion

The strawman algorithm in Section 3.1 computes auth-cookies by disabling combinations of cookies and determining whether a user remains logged in. Although this approach is simple and effective, it does not scale in practice, because

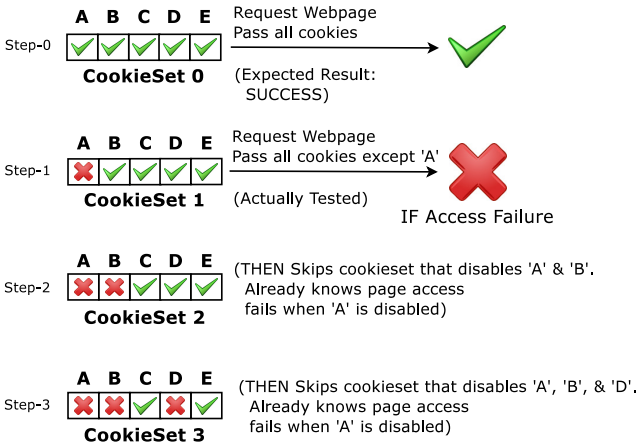


Figure 5: *Suppose that (A&B) is the auth-cookies combination. Newton does not know that. A page fetch request will not succeed if cookie A is not present in the cookie-set. From that result, Newton will conclude that A is part of auth-cookies combination. But, now it does not need to test other cookie-sets that do not contain A.*

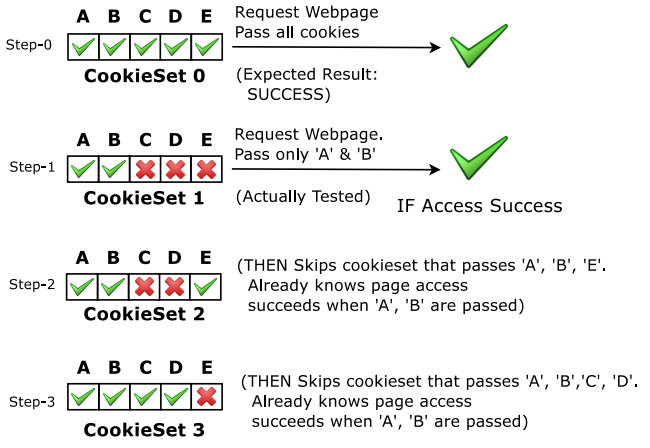


Figure 6: *Suppose that (A&B) is the auth-cookies combination. Newton does not know that. A page fetch will succeed because both A and B are included. From this, Newton concludes that either A, or B, or both are part of auth-cookies combinations. But, now it does not need to test other cookie-sets with A and B enabled.*

the number of tests is exponential in the number of cookies that a site sets, and many sites sent tens or hundreds of cookies; once the number of cookies on a site exceeds about 30, testing every combination becomes prohibitive. For example, `live.com` sets 27 cookies at login time. Assuming a single test requires about 200 ms, testing all combinations of only those cookies would require nearly a year. In this section, we discuss how to avoid the exponential cost of testing every possible cookie combination. Although the search space remains exponential, Newton can use knowledge from previous results to avoid testing certain combinations of cookies.

4.2.1 Basic optimization

First, Newton partitions the cookies for a site into two sets: "login cookies" (set during a user's login process) and "non-login cookies" (set either before or after the login event). To reduce testing time where possible, Newton initially assumes that a site's auth-cookies are a subset of the cookies that are set at login (the "login cookies"). As we discussed in Section 3.2, some auth-cookies may not be set at login, and Newton may need to expand the set of cookie-sets that it is testing to include additional cookies. Newton handles this case as well, but we describe the simple case first.

Figure 4 illustrates bounded partially ordered set representing all possible cookie-sets that Newton would have to test, either working downward from the top of the graph by adding cookies to an empty cookie-set, or by working upward from the bottom by removing cookies from a complete cookie-set. The Newton algorithm alternates by performing one test from the top of the partially ordered set, followed by one from the bottom, removing cookie-sets from the graph as they are determined to either represent an auth-cookie set or not. The algorithm terminates when no cookie-sets remain.

At first glance, it might appear that if a website sets N "login cookies", then Newton would need to send HTTP requests to the website with all 2^N possible cookie combinations to determine all sets of cookies that represent auth-cookies. Fortunately, we can use the outcome of some cookie-set tests to infer the outcomes for other cookie-sets, precluding the need to test all sets with HTTP requests. Figure 5 shows one such inference optimization: if Newton determines that the user is logged in when all cookies in the cookie-set are sent, but that the user is logged out when cookie A is not set, then Newton can conclude that A is part of an auth-cookies combination. Conversely, if Newton determines that a user is logged in when all cookies in a cookie-set are sent, and also that the user is logged in when *only* cookies A and B are sent, then Newton can infer that any cookie-set containing A and B will succeed, and either A or B or both of them are auth-cookies. Figure 6 shows a similar optimization.

4.2.2 Handling corner cases

User logout or session termination during test. When Newton is computing the auth-cookies for a particular site, the user's session may terminate for a variety of independent reasons; a user's session might time out, or a user might initiate a logout. If these events occur during one of Newton's tests, logout should not be attributed to set of cookies that were being tested at the time the logout occurred. For this reason, after the computation of the auth-cookies completes, Newton tests that the user is still logged in by sending all cookies for the site in an HTTP request. State V in Figures 7 and 8 illustrates this logic.

Finding auth-cookies that are not set at login. As we mentioned in Section 4.2.1, sometimes an auth-cookie might be set at some time other than when the user logs in, either because the site sets it at some later point after the user logs in (*e.g.*, as is the case with Google Calendar) or because the auth-cookie persisted since the user's previous session. There are two ways to detect that auth-cookies may be set at times other

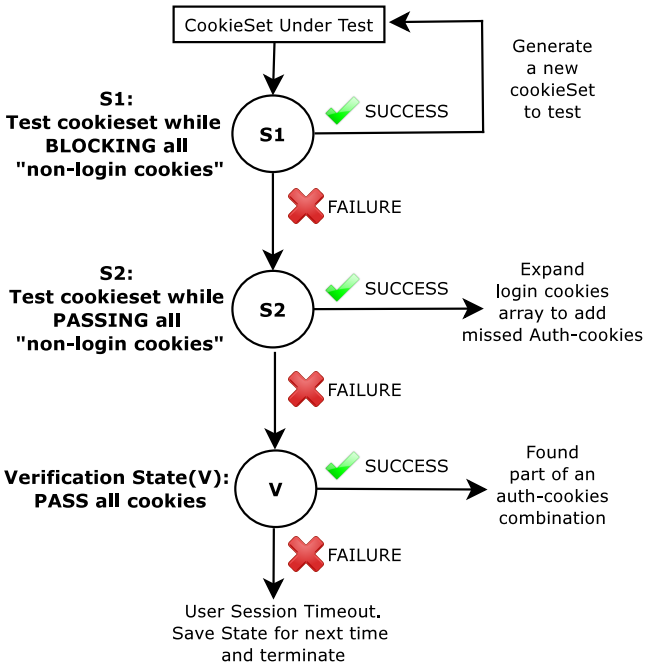


Figure 7: *When testing all current bottom cookie-sets(going UP in the POSET), if Newton discovers negative test result(testing a cookie-set still causes user to be logged in), then Newton moves on to next cookie-set to be tested. However, if the result is positive, then Newton performs two additional tests: one to ensure that there are no additional auth-cookies that are in non-login cookie set(state S2), and one to ensure that we have not mistakenly assumed a disabled cookies as an auth-cookie combination because a user was logged out during testing(state V).*

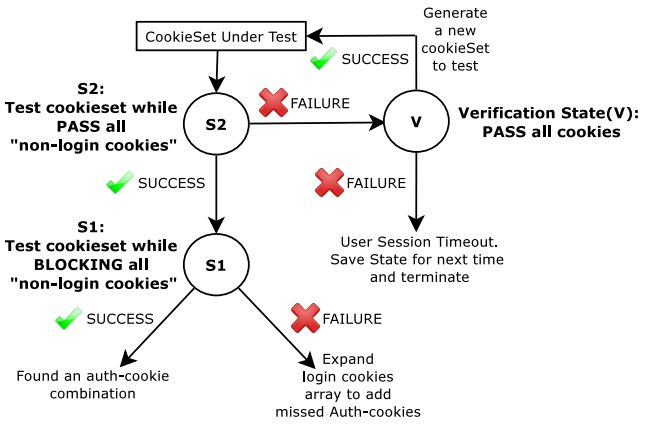


Figure 8: *In contrast to Figure-7, while testing all current tops(coming DOWN in the POSET), a negative test result occurs when user is not logged in(expected result with most of the cookies blocked) where as a positive test result is when a user is still logged in. To save unnecessary testing, we swapped the order of S1 and S2 to accommodate this test result role reversal. Once again state V is executed to ensure that previous negative result is not due to legitimate session log out.*

than when the user logs in; states *S*1 and *S*2 in Figures 7 and 8 show the logic to detect these corner cases. For example, in Figure 7, if a set of cookies from the "login cookies" set does not log a user in, but enabling all of the cookies that are not set at login results in a successful login, then Newton detects that one or more cookies that were not set at login must be part of the set of auth-cookies. In this case, Newton must actually proceed to compute the exact auth-cookies present in "non-login cookies" set. To do so, Newton constructs a partially ordered set of "non-login cookies" similar to the one shown in Figure 4 to find the additional auth-cookies.

5 Newton: Implementation and Performance Evaluation

We describe the implementation of Newton as a Chrome browser extension and our evaluation of its performance on real-world webpages and services.

5.1 Prototype: Chrome Browser Extension

We implemented Newton as a Chrome browser extension. Our code is open source, and interested security auditors can download, install it and start using with minimal configuration. Table 1 shows the breakdown of functions in terms of lines of code. Except for the scrapers of site-specific content, all of

Function	Lines of code
State Machine Driver	5,000
Cookie-set generation and optimization	1,700
State management and miscellaneous	1,300
Testing	800
Interpreter for scraping language (DSL)	1,600
Site specific content scrapers for 70 sites (DSL)	4,600
Total	15,000

Table 1: *Approximate lines of code to implement the Newton Chrome browser extension. All lines of code are Javascript except for the site-specific content scrapers, which are in Newton's domain specific language.*

the code was written in Javascript; we write the scrapers in a domain specific language.

A significant portion of the code resides in two modules. A major component of Newton's function is the *state machine driver*, which determines which elements in the partially ordered set from Figure 4 have been tested, how to generate the next tests based on previous results, and when to terminate and produce the set of auth-cookies. The state machine driver must also distinguish significant results from false positives that may arise due to network delay or other failures, as shown in Figures 7 and 8. To distinguish performance problems from login failures, Newton maintains a moving average of successful page load time. During a test, if a page fetch or load is not complete successfully within a factor of three of this time, Newton considers the result significant.

The *state management* logic copes with cases when a web session times out or user logs out in the middle of testing, Newton needs to periodically store the state of testing on a persistent device. This functionality is especially useful for

```
1  <div name="dropbox">
2  <action type="fetch-url">
3  https://www.dropbox.com/account#settings</action>
4    <div name="first-name" can_be_a_null="no">
5      <action type="store" field_type="editable">
6        td:contains("First name")+>input
7      </action>
8          </div>
9
10   <div name="last-name" can_be_a_null="no">
11     <action type="store" field_type="editable">
12       td:contains("Last name")+>input
13     </action>
14         </div>
15
16   <div name="email" can_be_a_null="no">
17     <action type="store"
18      jquery_filter="remove-children">
19       td:contains("Email")+
20     </action>
21         </div>
22 </div>
```

Figure 9: *Example of domain specific language for finding and a user's full name on Dropbox. For sites where we were able to create dummy accounts, we could write modules to infer the user's username or full name from the site, which would then allow Newton to determine if a user was logged in.*

web applications that have very short session time outs for security such as banking sites. However, the next time user logs in, the number of cookies created can be different from previous session. Our state management code takes care of resolving such differences.

Another significant component is the set of site-specific *scrapers* of usernames and URLs. The domain specific language that we implemented can traverse the user's account using directives to fetch a URL, follow a link, simulating a click and storing information. (We have implemented this tool to gather data for other studies, but for this paper we are only using the function involving username inference.) *All users who have installed the tool have consented to both the installation and the use of the data for the purposes of this study.* As previously mentioned, computations about a site's auth-cookies can be shared across users. To protect user privacy, however, we never associate a computation of a site's auth-cookies with a particular user (or username), and Newton always asks for the user's permission before uploading any data from the local machine.

Figure 9 shows an example of a scraper for `dropbox.com`. The code instructs the browser to go to the settings page for the user (line 3) and locate the HTML tags that correspond to the first and last name of the user (lines 5–14); it then stores the information to local storage (lines 16–21). We have shown one of the simpler examples of a scraper for simplicity; more examples are available on Github.[2]

5.2 Performance Evaluation

As we discussed, we envision that in addition to website administrators, users might also use Newton to detect when they are visiting vulnerable websites in real time. To evaluate the feasibility of this use case, we explore the number of page fetches, time, and amount of data required to compute the auth-cookies. The tests in this section were performed from a laptop on a residential network with downstream throughput of about 6–7 Mbps in the United States in 2014.

Figure 10a shows a distribution of the number of page fetches that are required to compute the auth-cookies for a particular site. Newton can compute auth-cookies for 90% of sites with fewer than 100 page fetches; the median number of page fetches to compute auth-cookies for a site is 25. Figure 10b shows that about 80% of these computations require less than four minutes to complete, and Figure 10c shows that about 80% of these computations require the client to download less than 20 MB of data from the webpage for which Newton is performing the auth-cookie computation. These results show that Newton is efficient enough to operate in practice. We have released Newton to a set of alpha users, who have been using the tool for about three months; none of these users have experienced any performance problems or disruptions as a result of running Newton. We will release Newton, as well as the source code for Newton and the data from the analysis in this paper when the paper is published.

Another possible concern is that Newton might introduce excessive server load. As shown in Figure 10a, Newton can compute auth-cookies for most sites with fewer than 100 HTTP requests; most web applications are designed to handle many more requests at much higher rates. We also note that the results of Newton's auth-cookie computations are not specific to each user, so if Newton shares the results of these computations across users, users can avoid redundant computation.

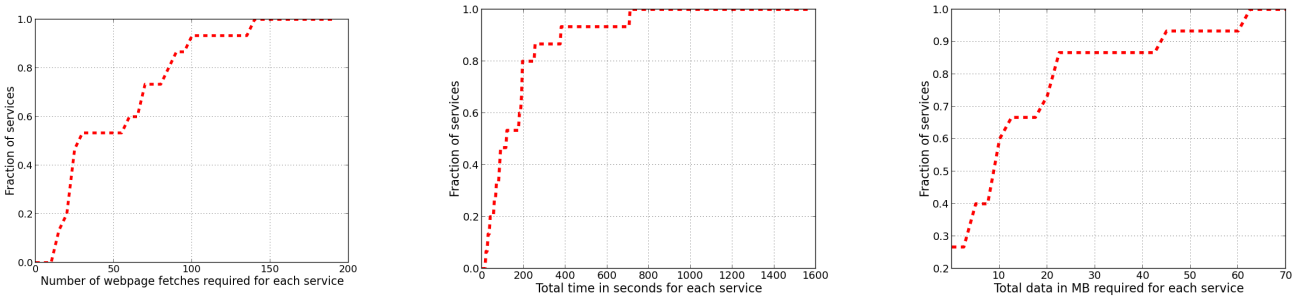
6 Case Studies and Recommendations

In this section, we use Newton to investigate how 45 different popular websites use auth-cookies for client authentication. Newton uncovered many design flaws and vulnerabilities, many of which could be exploitable in the context of the scenarios that we outline in Section 2. Based on these case studies, we develop a set of "best practice" recommendations for using auth-cookies for client authentication and also discuss the "cost" of implementing these recommendations for site operators. In some ways, this section represents a much-needed re-appraisal of the recommendations from Fu *et al.* more than ten years ago [13], when client authentication on the web was much simpler. In addition to the specific recommendations that we offer, we hope that users, web application developers, and website administrators can use the auditing functions and recommendations that Newton provides to improve the security of client authentication on today's websites.

6.1 Setting the `secure` flag

An attacker can gain access to user's private account if he is successful in stealing at least one complete auth-cookies com-

[2] We will make the location of the code public when the paper is published.



(a) *Number of page fetches.* **(b)** *Time.* **(c)** *Data.*

Figure 10: *Page fetches, time, and data required to compute auth-cookies.*

bination. As mentioned in Section 2, an attacker who controls hop in victim's network path (*e.g.*, as an open WiFi router, operated as a rogue, unscrupulous ISP, or other untrusted third party) can induce the victim's browser into divulging all cookies for which the `secure` flag is not set. For this reason, *we recommend that at least one auth-cookie in each auth-cookies combination should have the* `secure` *flag.*

Our investigation found nine sites that served at least some content over HTTPS, which would indicate that the auth-cookies were likely sensitive. *In four of those cases, the site always served content over HTTPS, but no cookie had the* `secure` *flag set.* In these cases, an on-path attacker could still induce the victim to divulge these presumably sensitive cookies over an insecure channel.

Another five sites served content over both HTTP and HTTPS; some auth-cookies had `secure` flag set, but others did not. Unfortunately, in these cases not enough auth-cookies had `secure` flag set to prevent theft of a complete auth-cookies combination. It is likely that these vulnerabilities exist because web programmers may not have easy ways to understand how different auth-cookie combinations grant control to different parts of a website. In the future, the use of a tool like Newton could help web programmers eradicate these types of errors.

We study two use cases where web programmers appear to have used the `secure` flag incorrectly:

GoDaddy. Newton discovered that `GoDaddy` has the following three auth-cookie combinations, with seven auth-cookies that grant access to a user's private account over HTTPS:

```
(ATL.SID.IDP AND gdCassCluster.F/cCGwbuE8)      OR
(ATL.SID.MYA AND gdCassCluster.osGjBIVhZQ       AND
                 MemAuthId1ANDShopperId1)       OR
                                  (auth_idp)
```

Of these, only `auth_idp` has the `secure` flag set to true. Thus, an attacker might steal any of the other six auth-cookies that satisfy either of the first two auth-cookie combinations to gain access to the user's account.

Yahoo. Newton found that two of the Yahoo's services have following auth-cookies combinations:

```
Yahoo Mail   :  Y AND T AND SSL
Yahoo Search :  Y AND T
```

Although both of these services serve content over HTTPS, only the SSL cookie is protected using the `secure` flag. This protects Yahoo Mail from cookie theft, but an attacker can nonetheless steal a second combination of cookies and still gain access to the complete past search history of the victim.

Implementing this recommendation: Setting the `secure` flag for auth-cookies of sites that are accessed entirely over HTTPS is easy. For sites that offer mixed content, developers must ensure that the auth-cookie combinations required for parts of site served over HTTPS are mutually exclusive from those required for HTTP parts of the site. Although implementing this recommendation still requires a thorough analysis of the website's services, we believe that this is more practical than HSTS deployment, which can only be implemented on a much more coarse granularity.

6.2 Setting the `HttpOnly` flag

An obvious mechanism for protecting the theft of auth-cookies via attacks such as XSS is to set the `HttpOnly` flag, but there may be legitimate reasons why a web programmer may not want to set the `HttpOnly` flag. For example, some Javascript programs may need to access a cookie. The programmer should always have one auth-cookie in each combination for which the `HttpOnly` flag is set. If a web programmer does not follow this practice, an XSS exploit or third-party malicious Javascript can gain access to the auth-cookies.

Newton found seven sites that serve at least part of the contents or all contents over HTTPS (suggesting that the content and the cookies should be considered sensitive), yet did not set the `HttpOnly` flag to protect all auth-cookie combinations. Of these seven vulnerable, it appears that

five of these vulnerabilities result from simple programming errors, since for those sites, *none* of the cookies had the `HttpOnly` flag. We explore one such use case below.

Amazon. We found that Amazon AWS has the following auth-cookie combination:

$$\begin{aligned}&(\texttt{aws-at-main AND aws-userInfo}\\&\texttt{AND aws-creds AND aws-x-main}) \quad \text{OR}\\&(\texttt{aws-at-main AND aws-userInfo AND}\\&\texttt{aws-creds AND}\\&\texttt{aws-ubid-main AND aws-session-id})\end{aligned}$$

In this case, both auth-cookie combinations are protected because `aws-creds` has `HttpOnly` set to true. Yet, combinations to access user's private account on Amazon's eCommerce site are as follows:

$$\begin{aligned}&(\texttt{at-main AND sess-at-main}\\&\texttt{AND ubid-main AND x-main}) \quad \text{OR}\\&(\texttt{at-main AND sess-at-main AND}\\&\texttt{ubid-main AND session-id})\end{aligned}$$

None of these cookies have the `HttpOnly` attribute set, leaving authentication to Amazon's eCommerce site vulnerable to cookie-stealing and authentication attacks.

Implementing this recommendation: A site operator can ensure that setting `HttpOnly` would not break site functionality by searching over the site's Javascript code to determine if whether any Javascript accesses the `HttpOnly` auth-cookie. In cases where all auth-cookies are accessed by Javascript code, the operator can secure the site by adding one more required auth-cookie to each combination that has the `HttpOnly` attribute.

6.3 Using auth-cookie entropy across sessions

At least one auth-cookie from each combination should change its value across the user's login sessions. If the auth-cookie values remain unchanged across different sessions, then a single theft of auth-cookies could enable an attacker to continue authenticating to a web service as the user for an indefinite amount of time. We found that six different sites did not change the client auth-cookies across sessions. Four of these sites also send auth-cookies in HTTP cleartext and are thus vulnerable to other attack vectors. We found Twitter and GoDaddy to be particularly vulnerable:

Twitter. `Twitter` has only one auth-cookie combination with a single auth-cookie: `auth_token`. The value of this cookie is 40 hex characters that is constant across sessions. Also because of this, if attacker steals this cookie, then she can login to victim's account even if victim is not logged in.

GoDaddy. In the case of GoDaddy, we observed that only `auth_idp` changes in its value across sessions. All other auth-cookies retain their value across sessions, meaning that even though `auth_idp` changes, other static auth-cookies could be used to re-authenticate the user even after a session has expired.

Some web applications change auth-cookies across different sessions, but the entropy of these changes is low. For example, Facebook has two auth-cookies, `c_user` & `xs`. The `c_user` cookie is a user ID that does not change. The `xs` cookie appears to change across sessions; this cookie has 36 characters that has multiple components, as follows: `119:ZHA1W3C7c7aBar:2:1406528127:6694`. Only the second part of this string is truly random; the other components are guessable, as they refer to characteristics such as the time when the user logs in.

Implementing this recommendation: The wide variety of cryptographic libraries in different web languages make it possible to use cryptographic primitives to introduce more entropy into session cookies; the challenge, of course, lies in selecting good inputs to these functions that serve as reasonable sources of entropy that are difficult to guess [1].

6.4 Invalidating auth-cookies upon logout

When a user explicitly initiates a log out action, it should invalidate the auth-cookies used in that session. If a site does not follow this practice, an attacker who can obtain the auth-cookies by the methods that we have discussed can authenticate as the user. If sites do not invalidate auth-cookies at logout, the logout action is essentially meaningless, and the user actually has no control over their sessions: a user may think he or she has logged out, but as long as the auth-cookies remain present and are not invalidated, the user is logged in. Newton found 18 sites where past session cookies allowed an attacker access to an account where the user was logged out.

If auth-cookies are not reset or invalidated when a user logs out, the cookies may also compromise a user's anonymity. Because the auth-cookie still carries the same value from the time when the user was logged in, all activity from the current anonymous context can be tracked to the user's real account on that site. Newton found five sites that do not modify their auth-cookies after the user has logged out, including Amazon, as described in the example below.

Amazon. After a user logs out of Amazon, both `session-id` and `ubid-main` retain their values from logged in session. Because each of them is a 17-digit number, each of them provides enough entropy to identify each user.

$$\begin{aligned}&(\texttt{at-main AND } \textit{sess-at-main} \texttt{ AND}\\&\texttt{ubid-main AND } \textit{x-main}) \quad \text{OR}\\&(\texttt{at-main AND}\texttt{sess-at-main}\textit{AND}\\&\texttt{ubid-main AND session-id})\end{aligned}$$

Implementing this recommendation: The difficulty of implementing this recommendation depends on the complexity of the website. Invalidating a user's auth-cookies upon logout requires deleting state about a user's past session. The growing popularity of NoSQL databases that provide only eventual (rather than strict) consistency, however, means that a implementing this feature correctly would require some care in these cases.

6.5 Changing auth-cookies when passwords change

We recommend that at least one auth-cookie in each combination should be derived from user's password in some cryptographic way such as HMAC. With this rule, even if user's password was leaked to an attacker and if she is using it to login to victim's account, as soon as victim realizes this and changes his password, all the auth-cookie combinations currently in use by attacker's session immediately become invalid. On eight sites (including Monster, Mailchimp, Quora, OKCupid, GoDaddy, Comcast, and Amazon), we obtained access to a user's account after changing the password, using old auth-cookies that were generated in a past session which was established after logging in using older password.[3] This vulnerability means that if a user's auth-cookies are compromised, an attacker can retain access to the user's account *even after the user discovers a compromise and changes his password!*

Implementing this recommendation: Existing websites already typically store user passwords as a cryptographic hash. This existing input could in turn be used to derive an auth-cookie that changes whenever the user's password changes.

7 Limitations

We discuss several limitations of Newton and possible areas for improvement.

7.1 Interfering with the User Experience

Some of Newton's tests of a website's practices depend on interactions with the user that should be as infrequent as possible or are not guaranteed to occur in the first place. For example, to test that a site invalidates a user's auth-cookies after the user has logged out, Newton would must discover that user has logged out, restore the previous session cookies, and test if the user is still logged in. Newton automatically discovers when a user logs out using various heuristics but to restore the auth-cookies, we explicitly ask for user permission, which become annoying if performed repeatedly. Currently we limit on how many such testing we conduct in a given time period to reduce this annoyance. Testing whether a website changes the user's auth-cookies when the user changes his or her password requires waiting for the user to first change his or her password. Currently, Newton automatically discovers when a user changes his or her password for a particular account, at which point it subsequently performs testing. Because Newton must wait until a user changes his or password to perform this test, we cannot guarantee that Newton always performs this test.

Because Newton retrieves webpages repeatedly while suppressing different combinations of cookies, a possible concern is that such repeated requests might have unintended or unforeseen side effects. An HTTP GET request is idempotent [12], so it should only have an effect the first time the user performs the request. It is possible that repeated requests might have unintended side-effects, however (*e.g.*, a user might be rate-limited if the client is perceived to be performing excessive requests), so we must be cognizant of this possibility when designing tests for specific sites.

7.2 Dealing with More Complex Websites

Every website is different, so Newton of course is not guaranteed to work flawlessly on every site. In this section, we mention certain instances where Newton did not work as expected. First, Newton decides whether a combination of auth-cookies results in a successful authentication based on whether the username is visible on the site. In some cases, however, we observed that if certain auth-cookies are missing, a web service may take more time to load (*e.g.*, Paypal attempts to reset missing auth-cookies) or wait for a user to confirm his or her identity (*e.g.*, Netflix asks for additional input). In these cases, Newton may erroneously conclude that the client failed to authenticate; in these cases, Newton may erroneously conclude that a certain cookie combination is an auth-cookie combination.

As we discussed in Section 4.2, because Newton considers the entire search space of cookie combinations, the tool has no false negatives. Yet, due to the absence of usernames on some pages during a test sequence, Newton might have false positives (*i.e.*, cases where a set of cookies are not in fact auth-cookies). In some other cases (*e.g.*, Stackoverflow), the page does not ever show a username, even if the user is logged in. Some of these cases can be solved with heuristics, such as waiting longer for the page to load or looking for certain features of the page other than the username—in some cases, however, manual auditing may be necessary.

Second, some sites such as Bank of America have a very large number of auth-cookies. Newton identified 28 of the auth-cookies on Bank of America but was not able to identify the complete set of auth-cookies in a reasonable amount of time. Even though, for this particular site, Newton could not completely enumerate all auth-cookies, Newton determined that the `SMSESSION` is necessary for authentication. In this case, `SMSESSION` has both the `secure` and `HttpOnly` attributes, so we were able to determine that Bank of America's auth-cookies were secure.

Finally, some sites have more complicated authentication mechanisms that can frustrate Newton's heuristics. For example, Newton assumes that all of a user's authentication tokens are client-side auth-cookies, but, in rare cases (*e.g.*, Grubhub), some sites maintain the user's login status on the server. In these cases, when the user logs out, the login state is maintained on the server side. Even though all cookies are still present, the user will be logged out, but Newton would mistakenly conclude that the user remains logged in. More complicated sites may also have multiple auth-cookie combinations, each of which determines different levels of site access. These more situations are corner cases today, but if

[3] We performed our tests of stale auth-cookies approximately ten minutes after changing the user password. It is possible that some of these sites may invalidate auth-cookies on a slower timescale.

these approaches become more common, future research may need to expand on Newton's techniques.

8 Related Work

At that time, only single authentication cookies were used; although many of the recommendations from that work are now commonplace, the rise of complex web applications and authentication mechanisms begs for a re-appraisal of client authentication. The Open Web Application Security Project (OWASP) has set security guidelines that recognize the importance of this problem and also acknowledge that simply setting `secure` and `HttpOnly` flags on cookies is not a viable means of securing auth-cookies for many web applications [25]. Other projects have explored both the security and privacy risks of exposed auth-cookies [7, 26, 30].

Existing browser-based security extensions attempt to protect a user's auth-cookies by setting the `HttpOnly` and `secure` flags and preventing these cookies from being accidentally passed to other domains [6, 8, 10, 22, 28]. All of these extensions rely on accurate mechanisms for computing the auth-cookies for a site; Newton can provide this capability. Various other systems (*e.g.*, SessionShield [22], ZAN [28]) have attempted to identify the cookies that serve as session identifiers, but, in contrast to Newton, none of these systems can identify auth-cookies. HTTPSEverywhere is a Chrome extension that forces traffic over HTTPS whenever possible [11], which may not always protect auth-cookies and may also hamper performance.

Concurrent work from Calzavara *et al.* has developed a supervised learning algorithm that attempts to identify auth-cookies for different sites [8]. The approach uses an approach similar to that which we describe in Section 4.2 to compute auth-cookies for user accounts in a controlled lab setting; using these sets of auth-cookies, the algorithm attempts to learn various features of auth-cookies to enable automatic detection of auth-cookies on other sites. The work differs from ours in several important ways.

First, the approach does not scale to general users and websites, because training the algorithm requires usernames and passwords, and because the algorithm can only estimate the auth-cookies of unknown websites. Unlike Newton, it cannot predict the exact auth-cookie combination for a previously unknown site when running on a user's machine. In contrast, Newton detects different auth-cookie combinations for different services in the same domain. Newton also computes auth-cookies "in the field" on arbitrary websites, rather than in the lab on much more limited set of sites. Second, Newton's performance optimizations allow it to run in the user's browser at runtime as a Chrome browser extension and compute auth-cookies for previously unseen websites, which allows it to gather more data from more users. Newton is thus both more scalable, more accurate, and more robust to changes in auth-cookie features that may occur over time.

Web applications can be sandboxed to prevent third-party Javascript libraries from leaking users' information [16, 29]; these systems solve a more general problem and require changes to underlying Javascript engines. Other approaches to add cryptographic properties to cookies [9, 13, 20] complement our recommendations.

Nearly 13 years ago, Fu *et al.* examined the use of cookies for client authentication on the web and uncovered various vulnerabilities; since this time, of course, the web has become increasingly complex [13].

9 Conclusion

The increasingly complicated nature of cookie-based authentication mechanisms on modern websites exposes clients to a new set of vulnerabilities. Websites have become so complex that it may be difficult for a user, website administrator, or web application programmer to determine which cookies are responsible for authenticating a client to different parts of a website, let alone whether a website's auth-cookie mechanisms are vulnerable to attack. Website designers and web application programmers need better tools to evaluate site security, as well as recommendations for best practices for securing these cookies. Towards these goals, we have developed a general algorithm to discover all auth-cookie combinations that will permit a user to access a site for *any* service on a particular website, including sub-services that are offered from the same website or domain (*e.g.*, Amazon Web Services vs. Amazon Shopping, Google Mail vs. YouTube). We have implemented this algorithm as a Chrome browser extension, Newton, which can discover auth-cookie combinations for a website without any prior knowledge about the website or the user. Our analysis of 45 popular websites revealed security vulnerabilities in auth-cookie combinations on 29 sites that could be exploited by relatively weak attackers. We used the case studies of vulnerabilities that we present to offer specific recommendations that could significantly improve website security with relatively little additional effort.

We are planning a public release of Newton; crowdsourcing data about auth-cookie mechanisms can ultimately help both users and web developers quickly identify and eradicate vulnerabilities in auth-cookies. We believe that Newton will be particularly useful to users who access the network via untrusted access points (*e.g.*, in hotels, coffee shops, and other public places). In these scenarios, Newton might ultimately take a more proactive role in protecting auth-cookies by ensuring that sensitive auth-cookies are not unnecessarily passed in the clear and possibly even proactively setting `secure` flags on sensitive cookies in the auth-cookie combinations. Newton could also be useful for helping users recover from attacks such as the recent Heartbleed attack [14], which allowed an attacker to steal auth-cookies even if the client sent them via HTTPS. As web authentication continues to become more complex, tools like Newton will increasingly valuable to users, web programmers, and website administrators.

References

[1] Advisory: Weak RNG in PHP session ID generation leads to session hijacking. http://seclists.org/fulldisclosure/2010/Mar/519. (Cited on page 12.)

[2] Unsafe cookies leave WordPress accounts open to hijacking, 2-factor bypass. Ars Technica. http://goo.gl/B1qLF7, May 2014. (Cited on page 1.)

[3] D. Atkins and R. Austein. *Threat Analysis of the Domain Name System (DNS)*. Internet Engineering Task Force, Aug. 2004. RFC 3833. (Cited on page 4.)

[4] A. Barth. *HTTP State Management Mechanism*. Internet Engineering Task Force, Apr. 2011. RFC 6265. (Cited on page 5.)

[5] A. Barth. *The Web Origin Concept*. Internet Engineering Task Force, Dec. 2011. RFC 6454. (Cited on page 3.)

[6] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. Automatic and robust client-side protection for cookie-based sessions. In *International Symposium on Engineering Secure Software and Systems*. ESSoS'14, 2014. (Cited on page 14.)

[7] E. Butler. A Firefox extension that demonstrates HTTP session hijacking attacks. http://codebutler.github.io/firesheep/, 2010. (Cited on page 14.)

[8] S. Calzavara, G. Tolomei, M. Bugliesi, and S. Orlando. Quite a mess in my cookie jar!: Leveraging machine learning to protect web authentication. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 189–200, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee. (Cited on page 14.)

[9] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. volume 12, pages 1:1–1:24, New York, NY, USA, July 2012. ACM. (Cited on page 14.)

[10] P. De Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: Self-reliant client-side protection against session fixation. In *Proceedings of the 12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS'12, pages 59–72, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on page 14.)

[11] HTTPS Everywhere. https://www.eff.org/https-everywhere. Electronic Frontier Foundation. (Cited on page 14.)

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hyptertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, June 1999. RFC 2616. (Cited on page 13.)

[13] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the Web. In *Proc. 10th USENIX Security Symposium*, Washington, DC, Aug. 2001. (Cited on pages 1, 10 and 14.)

[14] Hijacking user sessions with the Heartbleed vulnerability. https://www.mattslifebytes.com/?p=533. (Cited on page 14.)

[15] J. Hodges, C. Jackson, and A. Barth. *HTTP Strict Transport Security (HSTS)*. Internet Engineering Task Force, Nov. 2012. RFC 6797. (Cited on page 5.)

[16] L. Ingram and M. Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *USENIX Annual Technical Conference*. USENIX ATC 2012, 2012. (Cited on page 14.)

[17] Internet Explorer Web Platform Status and Roadmap. http://status.modern.ie/httpstricttransportsecurityhsts. (Cited on page 5.)

[18] ISP Advertisement Injection. http://zmhenkel.blogspot.com/2013/03/isp-advertisement-injection-cma.html. (Cited on page 4.)

[19] C. Jackson and A. Barth. Forcehttps: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 525–534, New York, NY, USA, 2008. ACM. (Cited on page 4.)

[20] A. Liu, J. Kovacs, C.-T. Huang, and M. Gouda. A secure cookie protocol. In *Proceedings of the 14th IEEE International Conference on Computer Communicati ons and Networks*. ICCCN-05, 2005. (Cited on page 14.)

[21] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 736–747, New York, NY, USA, 2012. ACM. (Cited on page 4.)

[22] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: Lightweight protection against session hijacking. In *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ESSoS'11, pages 87–100, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on page 14.)

[23] OWASP. OWASP Top 10 Application Security Risks - 2010. https://www.owasp.org/index.php/Top_10_2010-Main, 2010. (Cited on page 4.)

[24] OWASP: Double Submit Cookies. http://goo.gl/qmW7o5. (Cited on page 4.)

[25] Testing for cookies attributes. https://www.owasp.org/index.php/Testing_for_cookies_attributes_(OWASP-SM-002). (Cited on page 14.)

[26] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *9th USENIX Symposium on Networked Systems Design and Implementation*. NSDI 2012, 2012. (Cited on page 14.)

[27] SSL Pulse: Survey of the SSL Implementation of the Most Popular Web Sites. https://www.trustworthyinternet.org/ssl-pulse/. (Cited on page 5.)

[28] S. Tang, N. Dautenhahn, and S. T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 615–626, New York, NY, USA, 2011. ACM. (Cited on page 14.)

[29] J. Terrace, S. R. Beard, and N. P. K. Katta. Javascript in javascript (js.js): Sandboxing third-party scripts. In *Proceedings of USENIX Conference on Web Application Development*. WebApps '12, 2012. (Cited on page 14.)

[30] R. J. Walls, S. S. Clark, and B. N. Levine. Functional privacy or why cookies are better with milk. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security*, HotSec '12, Bellevue, WA, Aug. 2012. (Cited on page 14.)