Measuring Login Webpage Security

Steven Van Acker, Daniel Hausknecht, Andrei Sabelfeld
Chalmers University of Technology

ABSTRACT

Login webpages are the entry points into sensitive parts of web applications, dividing between public access to a website and private, user-specific, access to the website resources. As such, these entry points must be guarded with great care. A vast majority of today's websites relies on text-based username/password pairs for user authentication. While much prior research has focused on the strengths and weaknesses of textual passwords, this paper puts a spotlight on the security of the login webpages themselves. We conduct an empirical study of the Alexa top 100,000 pages to identify login pages and scrutinize their security. Our findings show several widely spread vulnerabilities, such as possibilities for password leaks to third parties and password eavesdropping on the network. They also show that only a scarce number of login pages deploy advanced security measures. Our findings on open-source web frameworks and content management systems confirm the lack of support against the login attacker. To ameliorate the problematic state of the art, we discuss measures to improve the security of login pages.

CCS Concepts

•**Security and privacy** → **Web application security;**

Keywords

web security; attacker models, login page, large-scale study

1. INTRODUCTION

Many websites on the Web today allow a visitor to create an account in order to provide a more personalized browsing experience.

Login as entry point To make use of their account on a website, users must authenticate to that website, typically by means of a login page. This authentication process separates the user experience in an unauthenticated and authenticated phase. This separation is crucial to the security and privacy of users and their information, because only the owner of an account is supposed to possess the correct credentials for logging in.

A malicious attacker with the ability to steal these credentials can impersonate the user and steal their private information, damage their image, cause financial losses or worse. Simply put, if the login page is insecure then the rest of the web application has no chance to be secure.

Attackers The setting of a login webpage demands a careful approach to modeling the attacker, as a combination of web and network attacker that attempts stealing login credentials. Hence, we focus on man-in-the-middle network attackers and third-party resource attackers. We consider browsers to be built securely on top of secure software libraries handling TLS, and also consider the server hosting the login page to be built equally securely and without vulnerabilities. We thus do not consider attacks on the browser software, such as drive-by-downloads or compromised web browsers, or attacks on the server software, such as SQL injection or remote code execution.

The presence of man-in-the-middle attackers on the Web is realistic, considering the availability of open and publicly available access points. In similar vein, considering the compromise of several trusted CAs [18, 16] in the past, it is not unrealistic to assume that a more powerful attacker has the ability to forge TLS certificates.

Nikiforakis et al. [30] point out that JavaScript code is often included from untrusted locations and that this code may be used to compromise the webpage in which the code was included. If a login page uses sensitive third-party resources such as JavaScript, Flash or even CSS, an attacker may compromise the server hosting these resources and compromise the login page this way. These third-party servers may even be malicious of their own with the desire to compromise login pages.

Once the credentials have been stolen, they can be leaked back to the attacker since browsers can not prevent the attacker from exfiltrating data [39].

Large-scale empirical study We examine how secure login pages are on the most popular 100,000 domains according to Alexa. The login page for a certain domain is located by looking for HTML input elements of the "password" type, by emulating the process in which a human would browse the website. Once located, we attack[1] the login page with five different attacker models and try to gain access to the password field. We find that 51,307 or 51.3% of the top 100,000 Alexa domains have a login page and that 32,221 or 62.8% of those login pages can be compromised by the most basic man-in-the-middle network attacker. Furthermore, we notice that the success rate of the attackers does not depend on the popularity of the domain, but that it remains fairly constant between the most popular and least popular domains of the Alexa top 100,000.

In our study, we are only interested in login pages implementing authentication mechanisms that exchange passwords

SAC 2017,April 03-07, 2017, Marrakech, Morocco
Copyright 2017 ACM 978-1-4503-4486-9/17/04...$15.00
http://dx.doi.org/10.1145/3019612.3019798

[1] No users or servers were affected by our attack experiments, see Section 4.1.3.

directly between a browser and a web application. We do not consider delegated authentication protocols like OAuth [31].

State-of-the-art support and suggested measures We consider that many web developers may build web sites based on popular web frameworks such as PHP or ASP.NET, or content management systems such as Wordpress or Drupal. We investigate the documentation of these web frameworks and CMSs to determine whether they give advice on the usage of any security mechanisms that help defend login pages.

Browser vendors have introduced and standardized several security mechanisms to combat these types of attackers. Unfortunately, we find that they are not widely used to secure login pages. We formulate recommendations on how these mechanisms can be combined in order to construct secure login pages.

Contributions The contributions made in this work are:

- We perform a large-scale empirical study on the Alexa top 100,000 domains to discover login pages and chart the usage of web authentication mechanisms. (Section 4)
- We perform a large-scale empirical study of the 51,307 previously discovered login pages to determine how they defend against the login attacker, by performing actual attacks on the login page to access the password-field. (Section 4)
- We study popular web frameworks and CMSs to determine what security precautions they advise in order to fend off attacks from the login attacker. (Section 5)
- Based on our examination of state-of-the-art security mechanisms implemented in browsers and their effect in stopping attacks from the login attacker, we formulate recommendations on how to build a secure login page. (Section 6)

2. BUILDING BLOCKS

Researchers and browser implementers developed different security mechanisms with the goal to mitigate certain attacks or to disable them completely. In this section we discuss those relevant to our attacker model. The attacker model itself is introduced in Section 3.

Mixed Content is a W3C standard that demands blocking requests over HTTP from within a webpage served over HTTPS [42]. The goal is to prevent attacks on insecure network connections introduced for example through including third-party content. Otherwise, this HTTP traffic can be modified by man-in-the-middle attackers which would ultimately put the main webpage at risk as well. The `block-all-mixed-content` (BAMC) CSP directive forces the Mixed Content mechanism to also block passive content such as images. Another CSP directive `upgrade-insecure-requests` [44] (UIR) automatically upgrades all HTTP requests to HTTPS.

Subresource Integrity (SRI) allows to detect potentially malicious modifications to resources by specifying the hash value of a resource. On loading, the hash value of the fetched resource is then matched against the specified value and an error is raised if the hash values do not match.

Even though SRI is a W3C candidate recommendation [43], it is so far implemented only by Firefox, Chrome (incl. mobile version), Opera and the Android browser [10].

HTTP Strict Transport Security (HSTS) forces future connections towards a hostname to be performed over HTTPS only. HSTS is standardized in RFC 6797 [1].

HSTS is enabled through a HTTP header coming with a server response over HTTPS. An attacker may have the chance to tamper with the very first connection attempt to a server, before the server has had the chance to activate HSTS. Therefore, major browser vendors maintain a HSTS preload list which is hard coded into the respective browser implementations [17]. HSTS is enabled by default for each domain in this list, ensuring that the browser will never try to connect to them via unencrypted HTTP. The HSTS preload list is not part of the standard, but is implemented by all major browser vendors.

HTTP Public Key Pinning (HPKP) is a HTTP header through which a certificate's public key can be associated with a hostname. This trust-on-first-use mechanism tries to reduce the problem where a certificate authority (CA) issues certificates for others than the actual domain owner, for example after a CA compromise. Such a certificate can then be used to, for example, launch a man-in-the-middle attack despite HTTPS. On establishing a HTTPS connection, the client's browser verifies the server's public key against the pin set during a previous connection and rejects the connection on mismatch. HPKP was standardized in RFC 7469 [2].

Since HPKP cannot protect clients against attacks on first connections, major browsers come with a hard coded list of trusted CAs for a specific domain [29]. This mechanism differs from the actual HPKP in that it pre-pins only the expected certificate issuing authorities, not the certificates themselves. We refer to the HPKP preload list as currently implemented by major browsers as the "indirect" HPKP preload list.

An alternative version of a HPKP preload list would store the public key of a host with the hostname, not the public key of the CA. This practice is called "end-entity pinning". Such a preload list would quickly become impractical and unmaintainable due to its size and maintainability requirements, Although not practical, this "direct" HPKP preload list is more secure because it cuts away the CA middle-man.

We will assume in the rest of this text that a "direct" HPKP preload list is implemented in the browser .

3. LOGIN ATTACKERS

The overall goal of our attackers is to steal user credentials from login pages, that is user names and passwords. To this end, the attacker tries to either passively sniff on network traffic or to actively inject malicious code into the webpage which then exfiltrates the desired information. We consider an attack as successful as soon as malicious code is successfully injected since browsers currently do not have reliable means to prevent data exfiltration [39].

We assume the client as well as the login page server are benign and that attackers cannot corrupt or control them through, for example, exploiting an implementation bug in

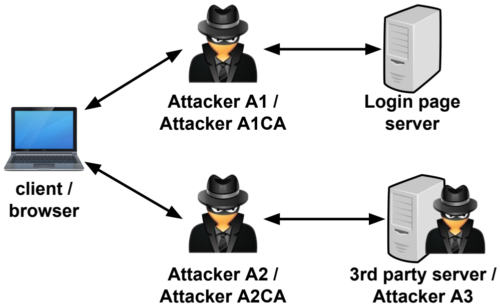

Figure 1: Attacker positions when a browser loads the login page from a server.

the client's web browser or an injection vulnerability. In line with related work [30], we consider any server reachable through a different domain name than the login page server as a third-party server even though it might actually be owned by the same person or company, for example different domains as part of a content delivery network.

We assume that attackers cannot break cryptographic primitives. In particular, communication over HTTPS is considered to be a secure channel which guarantees confidentiality and integrity of the transmitted messages. Discovering and fixing breaches in cryptographic implementations [28, 19, 34, 15] is a research field on its own, outside this paper's scope.

Based on these assumptions, we consider five different possible attackers as shown in Figure 1. We refer to this group of attackers as "login attackers", where each of the attackers is a "login attacker".

Attacker A1 The A1 attacker has the capabilities of an active network attacker as defined by Akhawe et al. [3] who can listen to and tamper with unencrypted traffic between client and login page server. In particular, A1 can inject malicious code and even disable any mitigation mechanism such as CSP or HSTS by simply removing the respective HTTP header. In case of an encrypted connection (HTTPS), if the client connects with the web server for the first time and its domain name is not registered with the client browser's build-in HSTS preload lists, the attacker can try to strip TLS to perform a man-in-the-middle attack [26].

Attacker A2 This attacker is almost identical to A1 except that A2 operates between the client and third-party servers. An attack by A2 can be prevented on the client side if the webpage uses SRI checks.

Attacker A3 The A3 attacker comes closest to the gadget attacker by Akhawe et al. [3] with the difference that we only consider third-party servers to dispatch malicious content. The attacker either directly controls the third-party server or has managed to corrupt a third-party server, opening up for possibilities to serve malicious resources, such as JavaScript, Flash or CSS files. Because A3 controls a communication endpoint, it is irrelevant if the resource transmission is encrypted or not. A3 attacks can be detected through SRI checks by the browser.

Attackers A1CA and A2CA Last, we consider a network attacker exactly as A1 and A2, respectively, but with the additional ability to have access to a certificate authority (CA). This covers for example the realistic scenario of governmental control or possible CA compromises. In either case, A1CA and A2CA can issue valid certificates and use them, for example, to launch man-in-the-middle attacks despite HTTPS. Therefore the attack vectors for A1CA and A2CA are the same as for A1 and A2 respectively, and the attackers can modify all transmissions. In case the server's domain name is also listed in the "direct" HPKP preload list of the browser, the forged certificate can be detected and the attack stopped.

4. EMPIRICAL STUDY OF ALEXA TOP 100,000 DOMAINS

As is common practice in large-scale studies, we performed an empirical study of the Alexa top[2] 100,000 domains in May 2016, to discover login pages and evaluate their level of security. This section describes the setup and results of this study. Code and other materials used during the experiment are available online [11].

4.1 Experiment setup

In general, this experiment consists of two pieces: finding the login page and then attacking it.

First, the login page for a given domain must be located because, for an automated tool, it is not always obvious where it is. Some login pages are on the front page of a website, while others can only be reached after following links and interacting with JavaScript menus.

Second, once the login page has been located, we emulate the different attacker models and attack the login page while it is visited. The goal of the attack is to steal the password that a user would enter on the website.

4.1.1 Login page assumptions

Mechanically locating the login page on a domain is a non-trivial task since they can require navigating the browser through e.g. JavaScript menu's and then be displayed in a foreign language and with custom styling. Because of these difficulties, we make a few assumptions about the general form of login pages based on sensible anecdotal observations and common sense.

First, we assume that webpages are written in HTML and that users authenticate via the webpage. Second, we assume that any login page has a password field and that this password field is an HTML "input" element of type "password". Third, if the login form is hidden under some JavaScript navigation menus, we assume that a user can properly navigate the menu structure in order to display the login form. Last, we only consider login pages that are hosted on the same domain, which we call "native" login pages. We do not consider a domain such as `youtube.com` or `blogger.com` to have their own login pages because they both redirect to the `google.com` login page, which is on a different domain.

4.1.2 Locating the login page

In order to find the login page on a given domain, we follow a couple of steps that we believe a sensible webpage visitor

[2] Obtained on 2016/03/26

would also follow. The search for a login page stops as soon as we have found a login page on the given domain. In this explanation, we will use the example domain `example.tld` to which the user wants to authenticate.

First, we visit the most-top level webpage of the domain using HTTP and HTTPS as if the user had typed it into the address bar of his browser. In this example, that would be `http://example.tld` and `https://example.tld`. In addition, we also try the same for `www.` prefix: `http://www-.example.tld` and `https://www.example.tld`.

Second, we retrieve all links from these four webpages and look for URLs that could lead to login pages, by filtering the URLs for login-related keywords in the top 10 most occurring natural languages [45] on the Web.

Third, we consult the search engine Bing and retrieve the domain's 20 most popular URLs by looking for "site:example.tld", and visit those URLs to look for a login form.

Fourth, we extract all links from the "Bing URLs" and like in the second step look for URLs to potential login pages.

Finally, if we still haven't found a login page, we point a custom crawler based on jÄk [33], a web crawler using dynamic analysis of client-side JavaScript to improve coverage of a web application, to the first working top-level URL in the domain and let it explore the website for up to 30 minutes looking for a login page.

Once a login page has been located, some data is gathered for statistics, as well as any necessary interactions with the webpage to get to the login form (in case of the jÄk crawler). This information can be used later to visit the page under the different attacker model scenarios.

4.1.3 Attacking the login page

Simply analyzing the login page and predicting whether it is safe based on implemented countermeasures, is not sufficient. Early experiments showed that certain JavaScript or Flash files related to web analytics were only briefly inserted in a webpage via JavaScript, and then promptly removed. This short lifetime of the resource on the webpage makes it difficult to detect using a passive analysis approach. To prevent a high false negative count, we opted to assume the role of an attacker and perform an actual attack on the login pages instead. Note that we do not attack the web servers in any way, only the web traffic towards the browser.

The attacks are fully automated for each of the attacker models and consist of two components: an automated web browser based on QT5's QWebView capable of rendering webpages and executing both JavaScript and Flash, and an HTTP proxy based on mitmproxy [35] v0.18 which simulated the attacker. To simulate the A1CA and A2CA attackers, we added mitmproxy's CA certificate to the certificate store used by the automated browser.

Attackers simulated via an HTTP proxy With the exception of A3, all attacker models are network-based, which motivates the use of an HTTP proxy to simulate the attacker. The A3 attacker model can equally be implemented at the proxy level, even though this attacker has compromised a third-party host instead of the network. The proxy can inspect all HTTP(S) requests and responses, but will only modify responses that are in "scope" for a specific attacker model. For instance, when assuming the role of the A1 attacker model, the proxy will only consider unencrypted requests that target the same domain as the login page to be in "scope".

As indicated in Section 2, several major browsers implement HSTS and HPKP. Unfortunately, our headless browser based on QT5's QWebView does not. We opted to build HSTS and HPKP awareness into the proxy, by interpreting the respective HTTP headers and acting upon them just like a normal browser would.

We refrain from using real browsers to perform the large-scale experiment since they are bulky in comparison with our headless browser. The advantage of a real browser supporting the latest security countermeasures is outweighed by the limited deployment of these countermeasures on visited login pages.

To prevent that the browser is redirected to an out-of-scope URL while retrieving a resource, we "hijack" the request by fetching the requested resource directly so that the browser never sees the redirect chain.

Finally, web servers may activate security measures by setting certain HTTP headers such as CSP, UIR, HSTS or HPKP. To avoid that these security measures become a problem in later HTTP requests, our proxy will remove them from any responses when those responses are in scope of the assumed attacker model.

Attack and payload On a login page, we consider HTML, JavaScript, CSS and Flash to be "sensitive" resources. To attack a login page, we therefore attack all sensitive resources in scope of and observed by a certain attacker model.

In identified HTML, JavaScript and Flash resources, we inject a piece of JavaScript that locates and reports accessible password fields in any parent- and sub-frames. This search is executed every second with `setInterval()`.

CSS resources are a special case because they do not contain any executable code. Unlike with HTML, JavaScript and Flash resources, stealing the password via a compromised CSS resource involves a non-trivial scriptless attack [21]. Instead of implementing such a scriptless attack, we only attempt to prove that a password field can be attacked using CSS, by tainting CSS values and checking for their presence in the rendered web page.

Automated browser visits the login page For a given domain, we revisit the previously identified login page through our proxy and replay any required JavaScript events, once for every attacker model. Each time, we wait up to one minute, after which any data reported by the attacker's JavaScript payload is retrieved and all password fields are examined for a CSS taint.

4.2 Results

We discovered native login pages on 51,307 or 51.3% of the top 100,000 Alexa domains. As explained in Section 4.1.1, keep in mind that we disregard login pages hosted on a different domain, and thus only consider "native" login pages.

Of the 51,307 discovered login pages, 48,547 (94.6%) could successfully be visited. We noticed that 27,238 (53.1%) login pages were served over, or eventually redirected to, HTTP and 21,309 (41.5%) over HTTPS. Of the 21,309 HTTPS login pages, 198 had an HTTP form target and would send the password unencrypted over the network upon submitting the password. In combination with those login pages served over HTTP, and without the need to perform actual attacks, we thus found that 27,436 (53.5%) login pages were already insecure because they either allowed content to be injected over an unencrypted connection, or they submitted the password over an unencrypted connection. For 5,761 (11.2%) login pages served over HTTP and 3,899 (7.6%) login pages served over HTTPS, we could not determine the target of the submission form, either because no enclosing HTML form was found or because the form submission was handled by JavaScript.

A total of 2,980 domains used HSTS of which 160 used it to disable HSTS by setting max-age to 0. As far as we could determine in our study in May 2016, no visited login pages used HPKP.

Out of 115 login pages that use BAMC, UIR or SRI, 4 use BAMC, 13 use UIR and 98 use SRI. Interestingly, no login page combined one of these technologies with another, so that these sets do not overlap.

	HTML	JS	CSS	SWF	Total
A1	28,346	24,116	21,021	768	30,945 (60.3%)
A2	176	16,057	4,592	724	16,452 (32.1%)
A3	159	35,759	10,039	984	36,031 (70.2%)
A[1,2]	28,372	27,581	23,245	1,460	32,221 (62.8%)
A[1,2,3]	28,411	40,164	27,868	1,902	42,284 (82.4%)
A1CA	40,054	35,975	32,732	889	43,799 (85.4%)
A2CA	284	28,716	9,890	1,025	29,404 (57.3%)
Total	41,672	43,200	38,666	2,196	45,968 (89.6%)

Table 1: Number of login pages compromised by each attacker model and the resource types they used for the successful compromise. A[. . .] denotes the combination of several attacker models. The percentage in the last column is calculated against the 51,307 domains with discovered login pages.

Table 1 summarizes the results of attacking the login pages with each of the attacker models, indicating how many login pages were successfully attacked per attacker model and through which resource type.

In total, the A1, A2, A3, A1CA and A2CA attackers managed to compromise 30,945 (60.3%), 16,452 (32.1%), 36,031 (70.2%), 43,799 (85.4%) and 29,404 (57.3%) login pages respectively. A network attacker able to man-in-the-middle all of the victim's traffic, denoted by A[1,2] in Table 1, is able to compromise 32,221 (62.8%) login pages. The combination of the classical attackers A1, A2 and A3, denoted by A[1,2,3], can compromise 42,284 (82.4%) login pages.

For JavaScript resources, it is striking how many domains include code from `google-analytics.com` and `facebook.net` on their login pages. The A3 attacker managed to compromise 26,016 (50.7%) and 10,483 (20.4%) login pages using code from these third-party domains respectively. Noteworthy is that eight out of top ten domains abused by A3 could not be attacked by either A2 or A2CA. These eight Google-owned third-party resource domains all appear on both the HSTS preload list and the "indirect" HPKP preload list, thus foiling these network attacks.

For Flash resources, all three attackers had the most success compromising login pages by injecting into Flash resources from `moatads.com`. This domain is listed as serving malware [41].

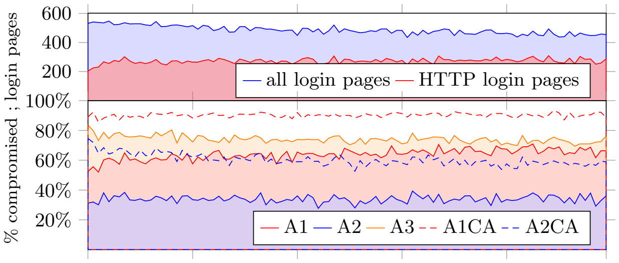

Figure 2: Evolution of the number of login pages (top) and success rates of the different attacker models (bottom) for decreasing domain popularity (in sets of 1000)

Figure 2 plots several metrics against the Alexa popularity rank of the domains in our study. In both plots, the horizontal axis indicates the domains sorted by decreasing popularity, in sets of 1000, with the most popular domains on the left-most side.

The top plot shows how many login pages we discovered and how many of those are served over HTTP. We observe here that we found more login pages among the more popular domains, and that login pages on popular domains are more frequently served over HTTPS.

The bottom plot in Figure 2 depicts the success rate, as a fraction of the discovered login pages, of the different attacker models against domain rank. The success rates for A2 and A1CA remain fairly constant and seem to be independent of domain rank.

4.3 Discussion

Our study shows that webform-based authentication is a very common authentication mechanism on the Web, with 51.3% of the top 100,000 domains having a "native" login page. A good amount of these can be easily compromised because they are either served over HTTP or submit the password over HTTP (60.3% of login pages can be compromised by the A1 attacker), or use third-party resources that can easily be compromised by a network attacker (32.1% of login pages compromised by the A2 attacker). In addition, 70.2% of login pages include third-party resources on their login page, placing a lot of trust in these third-party domains and allowing attacker A3 access to their users' passwords.

Powerful attackers with access to a CA can compromise a lot more login pages, 85.4% of login pages for A1CA and 57.3% for A2CA. The idea of attackers with these capabilities may seem outrageous, but it is realistic, as discussed in Section 1. Browser vendors are implementing defensive measures to protect against exactly this type of attacker. Good examples for their effectiveness are the third-party resources served by Google servers. Not only are these resources served over HTTPS, but the hosting servers employ both HSTS and HPKP and preload this information in the most popular web browsers. With these measures, the A1, A2, A1CA and A2CA attackers are effectively stopped.

Using third-party resources still requires a leap of faith, trusting that the organization hosting the resources does not turn malicious and starts serving malware that is then included in a login page. By using SRI, it can be ensured that a login page will not be compromised even if a third-party resource server becomes malicious. This defensive measure is already used by 98 login pages.

The data from this study shows that a lot of login pages are insecure, despite the existence of defensive measures that can help web developers to combat many types of attackers. In the next section, we look at the information available to web developers about how to create secure login pages.

5. STUDY OF WEB FRAMEWORKS

With 44.4% of all websites using a content management system (CMS) [46], support by web frameworks for secure login pages can largely impact the security of many websites. We perform a best-effort study of web frameworks and content management systems, collecting information about documentation or API support regarding setting up HTTPS connections, configuration of HTTP headers and educational material related to our threat model.

Based on the popularity indicated by BuildWith and W3Techs we selected various web frameworks [7, 47, 6] and CMSs [8]. We decided to ignore their classifications since there exists no clear line between web framework and CMS. But we assume an underlying webserver with features comparable to Apache or nginx, for example the ability to set up HTTPS.

All studied web frameworks, e.g. PHP, ASP.NET, Java EE or Django, provide an API for defining any HTTP headers. This means, even without access to the underlying server, an application developer can set security related HTTP headers. We were not able to find security related educational material for all web frameworks. That is only for ColdFusion, Ruby on Rails, Express.js and Django, but not for PHP, ASP.NET, Java EE and Laravel. Interestingly, Java EE and Django both implement a feature to ensure HTTPS connections through the respective framework.

We could not find any CMS which documents a feature to set HTTP headers directly through the system itself. We interpret this that all CMSs rely on the underlying web frameworks to provide this functionality. We were not able to find security related educational material for all CMSs. That is only for Drupal, Joomla, TYPO3, Craft and Mura, but not for Wordpress, DNN Software, Umbraco, Concrete5 and Plone. It must be noted however that for frameworks with a plugin system, e.g. Wordpress, there are many security related plugins available. Though we do not discuss them here, these plugins often do provide security information. For several CMSs (Wordpress, Drupal, DNN Software, Umbraco, Craft) we found the configuration option to ensure HTTPS connections with a web application.

Our findings show that despite their popularity, the support for security measures, either in the form of documentation or directly through APIs, is not always provided. We argue that even though framework developers cannot predict how their software is used they should be more consequent in creating the awareness for security issues and should directly facilitate the configuration of HTTPS and security related HTTP headers to reduce efforts for non-security experts.

6. SECURITY RECOMMENDATIONS

Table 2 summarizes the effectiveness of security countermeasures in the context of each of the five attacker models. We discriminate not just between HTTP and HTTPS requests, but also whether or not the browser is sending a request to a domain for the first time.

From this table, it is clear that the different attackers, except for A3, can be stopped through the use of HTTPS in combination with preloaded HSTS and preloaded HPKP for all network resources including the login page itself. A3 attacker can be stopped through SRI for any resources.

In Section 2 we differentiate between an “indirect” and a “direct” HPKP preload list. With an “indirect” HPKP preload list, it is possible for a powerful attacker to compromise a CA on the mentioned whitelist and manage to forge a trusted certificate. With a “direct” HPKP preload list, there is no intermediate CA that can be compromised, but the downside is that the preload list becomes costly to maintain.

Both versions have disadvantages, but are better than not using HPKP or trust-on-first-use HPKP. Definitively solving the “rogue CA” problem is the focus of ongoing research in other fields, briefly summarized in work by Kranch et al. [23]

At this time, full protection is currently impractical since not all browser vendors support all security measures yet. However, as noted in Section 2, these security measures are on the standardization track and it is only a matter of time before they are adopted by all browser vendors.

7. RELATED WORK

To the best of our knowledge, we are the first to conduct a large scale empirical study in which login pages are automatically identified and analyzed for security measures. In this section, we discuss other research related to our work.

Empirical studies on webpage security Other researchers have analyzed the security of web sites, with the focus on a specific security measure [50, 51], a specific geographic origin [12, 40] or a specific browser technology [22]. Wang et al. manually investigated 188 login pages, examine whether the password was submitted in clear text and then build a browser extension based on their findings [49]. Kranch et al. [23] study HSTS and HPKP deployment and configurations in depth for domains on the respective preload lists, and shallowly for the Alexa top one million. They find that the adoption of these security measures is low, often misconfigured and often leak cookie values. Chen et al. [13] perform a large-scale study of mixed-content websites on the HTTPS websites in the Alexa top 100,000. They find that 43% of them make use of mixed content and list some examples of affected security-critical mixed-content webpages. Our focus is on the security of the password field on login pages in the Alexa top 100,000 domains, which we systematically and mechanically discover and evaluate against several real-

	HTTP										HTTPS									
	first request					next requests					first request					next requests				
	A1	A2	A3	A1	A2	A1	A2	A3	A1	A2	A1	A2	A3	A1	A2	A1	A2	A3	A1	A2
				CA	CA				CA	CA				CA	CA				CA	CA
SRI		✓	✓		✓		✓	✓		✓	✓	✓	✓	⋆	✓	✓	✓	✓	⋆	✓
HPKP											✓	✓		⋆	⋆	✓	✓		✓	✓
pre-HPKP											✓	✓		✓	✓	✓	✓		✓	✓
HSTS						✓	✓		⋆	⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆
pre-HSTS	✓	✓		⋆	⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆
BAMC		✓			⋆		✓			⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆
UIR		✓			⋆		✓			⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆

Table 2: Which countermeasures offer protection against which attacker models, for first and subsequent requests sent over HTTP and HTTPS. A check mark indicates successful protection, ⋆ indicates protection in case the remote server's public key is preloaded in the browser (pre-HPKP means "direct" HPKP preload list)

world attackers.

Third-party content Prior work has analyzed potentially malicious third-party content. Nikiforakis et al. [30] report on a large-scale empirical study on how web applications include third-party JavaScript code and discuss the issue of self-hosting of libraries as opposed to dynamically linking to third-party domains. Li et al. [25] study the threat of online advertising and the identification of sources serving malicious advertising. Rydstedt et al. [36] analyzed popular web sites with respect to defenses against frame busting techniques. Lekies et al. [24] research the problem of malicious content caching in web browsers and its practicality through a study of the top 500.000 Alexa domains. Canali et al. [9] take an opposite approach by developing a filter for web crawlers which identifies benign webpages such that they can be excluded from further analysis. Orthogonal to those works, we do not try to identify malicious content on the Web but study the implementation of security measurements on login pages and the level of protection they provide for these pages.

Framework analysis Meike et al. [27] analyze two open-source content management systems with respect to their security features, but put their focus on different attacks. Heiderich et al. [20] analyze client-side JavaScript-based web frameworks for security features such as sandboxing mechanisms and provides code samples to attack the frameworks. Their work is complementary to ours since also here another attacker model is considered.

Web app security recommendations The Open Web Application Security Project (OWASP) maintains a collection of existing security technologies and guidelines for webserver and web-client security, the OWASP Cheat Sheets [32].

Password security There exist numerous works on the strength of a password, e.g. [14, 5, 4, 48]. In Section 3, we defined the goal of our attackers to steal user names and passwords from login pages. Therefore password strength does not affect the success of an attacker in our model. Other password related works analyze special cases under our set-up. For example, Stock et al. [37] analyzes password managers and their ineffectiveness to protect passwords after a successful code injection attack. Van Acker et al. [38] investigate password meters and generators and possible password stealing attacks imposed through malicious third party services.

8. CONCLUSION

Login pages are of crucial importance to the security and privacy of web users' private information, because they handle a user's login credentials. In this work, we evaluate the security of login pages against a login attacker model, which encompasses man-in-the-middle network attackers with and without certificate-signing capability from a trusted certificate authority, as well as a third-party resource attacker. By performing actual attacks against the 51,307 login pages we discovered in the Alexa top 100,000, 32,221 or 62.8% of login pages can be compromised fairly easily by a man-in-the-middle attacker without special certificate-signing privileges. The fraction of login pages which can be compromised is independent of the domain's popularity rank. We evaluate existing browser security mechanisms designed to counter our different attacker models and conclude that today's browsers implement the needed security tools to offer end-users a secure login page. However, a study of the most popular web frameworks and CMSs reveals that information on how to build a secure login page, is not always available to web developers. Finally, we discuss measures and best practices to improve the security of login pages.

Acknowledgments This work was partly funded by Andrei Sabelfeld's Google Faculty Research Award, Facebook's Research and Academic Relations Program Gift, the European Community under the ProSecuToR project, and the Swedish research agency VR.

9. REFERENCES

[1] RFC 6797: HTTP Strict Transport Security (HSTS).

[2] RFC 7469: Public Key Pinning Extension for HTTP.

[3] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J. C., and Song, D. Towards a Formal Foundation of Web Security. In *CSF* (2010).

[4] Al-Ameen, M. N., Fatema, K., Wright, M. K., and Scielzo, S. Leveraging Real-Life Facts to Make Random Passwords More Memorable. In *ESORICS* (2015).

[5] Blocki, J., Datta, A., and Bonneau, J. Differentially Private Password Frequency Lists.

[6] BuiltWith. Framework usage statistics. http://trends.builtwith.com/framework.

[7] BuiltWith. Programming language usage. http://trends.builtwith.com/framework/

programming-language.

[8] BuiltWith. Statistics for websites using open source technologies. http://trends.builtwith.com/cms/open-source.

[9] Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *WWW* (2011).

[10] caniuse.com. Subresource Integrity. http://caniuse.com/#feat=subresource-integrity.

[11] Chalmers CSE. Related materials. http://www.cse.chalmers.se/research/group/security/measuring-login-page-security.

[12] Chen, P., Nikiforakis, N., Desmet, L., and Huygens, C. Security Analysis of the Chinese Web: How Well is It Protected? In *CCS SafeConfig* (2014).

[13] Chen, P., Nikiforakis, N., Huygens, C., and Desmet, L. A dangerous mix: Large-scale analysis of mixed-content websites. In *ISC* (2013).

[14] Dell'Amico, M., and Filippone, M. Monte Carlo Strength Evaluation: Fast and Reliable Password Checking. In *CCS* (2015).

[15] DROWN. CVE-2016-0800.

[16] Fisher, D. Final Report on DigiNotar Hack Shows Total Compromise of CA Servers. https://threatpost.com/final-report-diginotar-hack-\ shows-total-compromise-ca-servers-103112/77170/.

[17] Google Chrome. HSTS Preload Submission. https://hstspreload.appspot.com/.

[18] Group, C. Comodo SSL Affiliate The Recent RA Compromise. https://blog.comodo.com/other/the-recent-ra-compromise/.

[19] Heartbleed. CVE-2014-0160.

[20] Heiderich, M. Mustache security. https://code.google.com/archive/p/mustache-security/.

[21] Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: stealing the pie without touching the sill. In *CCS* (2012).

[22] Kontaxis, G., Antoniades, D., Polakis, I., and Markatos, E. P. An Empirical Study on the Security of Cross-domain Policies in Rich Internet Applications. In *EUROSEC* (2011).

[23] Kranch, M., and Bonneau, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS* (2015).

[24] Lekies, S., and Johns, M. Lightweight Integrity Protection for Web Storage-driven Content Caching. In *W2SP* (2012).

[25] Li, Z., Zhang, K., Xie, Y., Yu, F., and Wang, X. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *CCS* (2012).

[26] Marlinspike, M. sslstrip. http://www.thoughtcrime.org/software/sslstrip/.

[27] Meike, M., Sametinger, J., and Wiesauer, A. Security in open source web content management systems. *S&P* (2009).

[28] Meyer, C., and Schwenk, J. SoK: Lessons Learned from SSL/TLS Attacks. In *WISA* (2013).

[29] Mozilla. Public Key Pinning. https://wiki.mozilla.org/SecurityEngineering/Public_Key_Pinning.

[30] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *CCS* (2012).

[31] OAuth. OAuth. http://oauth.net/.

[32] OWASP. Cheat sheet series. https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series.

[33] Pellegrino, G., Tschürtz, C., Bodden, E., and Rossow, C. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID* (2015).

[34] POODLE. CVE-2014-3566.

[35] Project, M. mitmproxy. https://mitmproxy.org/.

[36] Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP* (2010).

[37] Stock, B., and Johns, M. Protecting Users Against XSS-based Password Manager Abuse. In *ASIACCS* (2014).

[38] Van Acker, S., Hausknecht, D., Joosen, W., and Sabelfeld, A. Password Meters and Generators on the Web: From Large-Scale Empirical Study to Getting It Right. In *CODASPY* (2015).

[39] Van Acker, S., Hausknecht, D., and Sabelfeld, A. Data Exfiltration in the Face of CSP. In *AsiaCCS* (2016).

[40] van Goethem, T., Chen, P., Nikiforakis, N., Desmet, L., and Joosen, W. Large-Scale Security Analysis of the Web: Challenges and Findings. In *TRUST* (2014).

[41] VirusTotal. js.moatads.com domain information. https://www.virustotal.com/en/domain/js.moatads.com/information/.

[42] W3C. Mixed Content. https://www.w3.org/TR/mixed-content/.

[43] W3C. Subresource Integrity. https://www.w3.org/TR/SRI/.

[44] W3C. Upgrade Insecure Requests. https://www.w3.org/TR/upgrade-insecure-requests/.

[45] W3techs. Usage of content languages for websites. http://w3techs.com/technologies/overview/content_language/all.

[46] W3techs. Usage of content management systems for websites. http://w3techs.com/technologies/overview/content_management/all.

[47] W3techs. Usage of server-side programming languages for websites. http://w3techs.com/technologies/overview/programming_language/all.

[48] Wang, D., and Wang, P. *The Emperor's New Password Creation Policies.* 2015.

[49] Wang, X. S., Choffnes, D., Gage Kelley, P., Greenstein, B., and Wetherall, D. Measuring and Predicting Web Login Safety. In *W-MUST* (2011).

[50] Weissbacher, M., Lauinger, T., and Robertson, W. K. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID* (2014).

[51] Zhou, Y., and Evans, D. Why aren't HTTP-only cookies more widely deployed. In *W2SP* (2010).