# The Password Problem

Elizabeth Walkup
Stanford University

Passwords are an ubiquitous, established part of the Internet today, but they are also a huge security headache. Single sign-on, OAuth, and password managers are some of the solutions to this problem. OAuth is a new, popular method that allows people to use large, common authentication providers for many web applications. However, it comes at the expense of some privacy: OAuth makes users easy to track across websites, applications, and devices. Password managers put the power in the hands of the users, but this vulnerability survey reveals that you have to be extremely careful which program you choose. All in all, password managers are the solution of choice for home users and small organizations, but large companies will probably want to invest in their own SSO solutions.

Ø

## 1. INTRODUCTION

Passwords are an ubiquitous, established part of the Internet today, but they are also a huge security headache. The average user has so many different accounts with different sites that it's almost impossible to remember them all. This leads to problems like password reuse - according to a 2012 study, over 61% of people reuse passwords on multiple sites [1]. This is a problem because if one site account gets stolen, hackers will test that username and password on other sites as well. People will also write down passwords on paper or in text files (which makes them easier to steal). And in general, people tend to choose weak passwords on their own, especially predictable ones that are vulnerable to dictionary attacks. Several methods have evolved to help combat this issue: password managers, single sign-on (SSO), and two-factor authentication. This papers examines these methods and the protections they offer against different attackers.

### 1.1 Single Sign-On (SSO)

SSO is an authentication process that allows users to have one account (and so one password) to access multiple different applications. Traditionally, these have been used internally in companies, where the applications are all owned by the same entity. There are many flavors of SSO, since organizations like to implement their own version. Some common SSO protocols include SAML, CoSign, CAS, and WebAuth, which is used here at Stanford.

A new method similar to SSO has arrived on the scene recently: OAuth. It is not quite the same as SSO - users must enter their username and password for each application. But it is the same username and password for each one, which cuts down on the number of accounts a user must remember. OAuth by itself is an authorization protocol, not an authentication protocol, but it is convenient because it allows applications to use the outside authentication provider of their choice. OAuth is also the basis for authentication protocols like OpenID Connect and Facebook Connect. Any time you log into somewhere using your Facebook or Google account, you are using OAuth. This convenience has gained OAuth a huge following, especially among mobile apps and small web apps. Because of its popularity and recentness, this paper will focus on OAuth more than traditional SSO protocols in Section 3. Note that OAuth has had two major iterations: OAuth 1.0 and OAuth 2.0. They are very different protocols - this paper will only cover OAuth 2.0, which is the one that is currently popular.

### 1.2 Password Managers

Password managers are a familiar concept to most people: they store all your passwords in an encrypted vault, so the only password you have to remember is the master password for the vault. However, over the years, the scope and platform of the average password manager has changed dramatically. Now, almost every password manager is a desktop app linked with browser extensions linked to a cloud storage account, possibly also linked to multiple devices and mobile apps. This is more convenient for users - passwords can be accessed from anywhere and filled in without typing. But it greatly increases the attack surface of the password manager program, which becomes a single point of failure in user security. And often, these components cannot be separated easily - for instance, the Dashlane program will not let you use the desktop app until you install their browser extension (although you can uninstall it later). 90% of password managers come with browser extensions and mobile applications, and 60% offer cloud accounts. For a full list of all the platforms offered by the examined password managers, see Appendix A. In Section 4, this paper will examine the new security benefits offered by modern password managers as well as their vulnerabilities.

### 1.3 Two-Factor Authentication

There are three things that can be used for authentication: something you know, something you have, and something you are. Passwords are something you know, but, as mentioned, they are often stolen or cracked. Two-factor authentication addresses the security of passwords without changing the number of the passwords a user has to remember.

In two-factor authentication, the user must supply a second form of authentication in addition to the normal password. It could be something they have, like a keyfile fob, or a code from a mobile device, or it could be something they are, like fingerprint reading. This means an attacker attempting to access the account cannot get in with just the password - they must get around the second factor as well, which can be difficult, especially for a remote attacker with no physical access to the target user.

Two-factor authentication can be used both with normal passwords, SSO, and with password managers for extra security. The most commonly used second factor is something you have - a phone. A one-time code is sent via text or voicemail, or obtained through an app. For instance, Stanford's WebAuth uses Duo, a mobile app, to supply the second factor when logging in. All major OAuth providers, including Facebook, Google, Twitter, Tumblr, and Github, have a phone-based two-factor option. However, none of them use it by default - the user must opt in. Google also allows users to use a USB with a security key, but in general this is not supported.

Of the password managers surveyed, 80% had support for two-factor authentication in the form of mobile device codes, similar

to SSO. Hoever, password managers also commonly use biometrics as a second factor, especially fingerprints, as more and more computers and phones come with readers built in. 60% had support for biometric two-factor authentication - all of these offered fingerprint support, and 10% offered the less secure face recognition option as well (LogMeOnce and True Key). 30% supported the use of an external USB key (most commonly YubiKey) as a second factor - these were Dashlane, KeePass, Keychain Access, LastPass, LogMeOnce, and PassPack. For the full list of password managers that support two-factor authentication, see Appendix B.

### 1.4 Attacker Types

Four different kinds of attackers will be considered when discussing security:

(1) Passive network attacker - this attacker can view, but not modify, network traffic between the victim and the server. They may also attack the server to steal information.

(2) Active network attacker - can both view and modify network traffic between the victim and the server. They can also intercept traffic, allowing them to so things like perform man-in-the-middle attacks or SSL downgrade attacks.

(3) Malware attacker - this attacker has malware installed on the victim's machine and is looking to exfiltrate more valuable information. Malware actions include keylogging, intercepting network traffic, or checking the clipboard contents.

(4) Physical attacker - this attacker has physical access to both the victim and their computer. They could install malware, run cracking programs, use the computer, or find passwords written down nearby.

## 2. RELATED WORK

### 2.1 OAuth

In 2012, J. Bonneau et al.[10] recognized the growing need to replace passwords and did a large survey of OAuth, single sign-on, password managers, and other solutions available. They found that, in general, single sign-on methods seemed to offer the best usability and security improvements. However, they only looked at two password managers: Firefox and LastPass. However, people like Sun and Beznosov[9] were already looking critically at OAuth 2.0: they found several vulnerabilities in the provider implementations of Google, Facebook, and Microsoft, though they found none in the protocol itself. Other OAuth vulnerability discoveries include that of Andrey Labunets[11], who used redirect URI vulnerabilities and RPC calls to leak information and attack provider websites. Even years later, OAuth implementations are still not immune to attacks, as demonstrated by the work of P.Hu and W. C.Lau[12], which used app impersonation to gain access to a huge amount of data in Facebook's social graph.

### 2.2 Password Managers

There have been several studies done on vulnerabilities in various parts of password managers. In 2014, Silver et al. [6] studied autofill vulnerabilities, attacks, and defenses in both desktop and mobile password managers. Some of the research done on password managers in this paper extends on their work. Li, Zhiwei et al. [5] They analyzed 5 of the most popular web-based password managers, evaluating them in 4 areas: bookmarklet vulnerabilities, classic web vulnerabilities, logic vulnerabilities, and UI vulnerabilities. They found that 4 out of the 5 password managers had vulnerabilities that allowed arbitrary password stealing (in light of a network-based attacker). Since bookmarklets have largely fallen out of favor now, this new research looks at a larger variety of password managers and their web vulnerabilities.

P. Gasti and K. B. Rasmussen[7] have taken a crack at breaking the database encryption methods of password manager vaults, while Chatterjee et al.[8] proposed a password manager called NoCrack that generates plausible but fake passwords when given the wrong master password. The idea is to make it take much longer for attackers to brute force a stolen password database.

## 3. SINGLE SIGN-ON (SSO)

### 3.1 The OAuth Protocol

OAuth 2.0 is more of a framework than a protocol - it leaves many implementation details, such as data storage, up to the programs that use it. It provides authorization but not authentication - an important distinction that often confuses people. An outside provider must supply authentication, or use a protocol like Facebook Connect or OpenID, which are authentication built off of OAuth. The goal of OAuth is to allow applications to access data on a server without giving the app the user's credentials. This means the user does not have to create a separate account for the app, and their credentials are not stored or given away needlessly.

There are four types of authorization flows:

(1) Authorization Code Flow - the user is redirected to the authentication server, where they enter their credentials and authorize the app to access their data. The server sends the app an authorization code, which it then trades for an access token.

(2) Implicit Flow- the user is redirected to the authentication server, where they enter their credentials and authorize the app to access their data. The server sends the app an access token to their redirect URI.

(3) Client Credentials Flow - a POST is done directly to the authentication server from the app using credentials belonging to the client app. The server sends an access token back.

(4) Resource Owner Password Flow - a POST is done directly to the authentication server from the app using credentials belonging to the user. The server sends an access token back.

The Authorization Code flow is by far the most widely used. Client Credentials flow is used for applications that do not use user interaction.

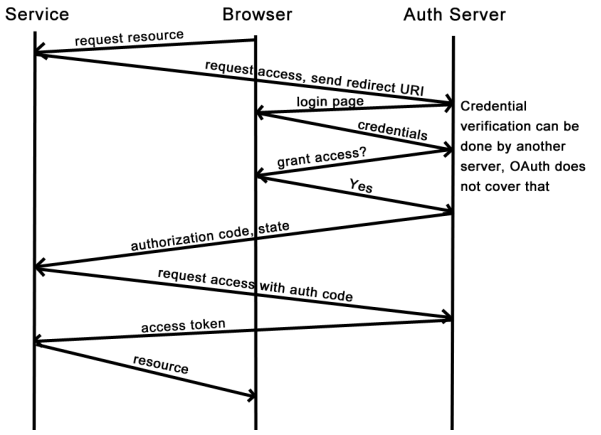

Fig. 1: OAuth Auth Code Flow

## 3.2 OAuth Implementations

OAuth has benefited from being a younger protocol by taking full advantage of open-source and hobby coding to build up a large set of implementations. There are over 50 OAuth implementations in 12 different programming languages. However, there are some problems.

For starters, a good chunk of OAuth implementations (roughly 20%) are for OAuth 1.0, which is no longer in common use. There is even a confusingly-named Python project called python-oauth2 that only implements OAuth 1.0. Most maintained projects will implement both, but it can be confusing sorting through them all. For the purposes of this research, only OAuth 2.0 implementations will be considered.

The confusion is not just present among open source users - OAuth provider companies are also a mixed bag. 41% of the top 61 providers still use OAuth 1.0 [14]. 11% (including LinkedIn, Twitter, and Google) implement both OAuth 1.0 and OAuth 2.0, mostly for backwards compatibility, since OAuth 1.0 cannot be used directly with OAuth 2.0.

As is often the case in open-source projects, they were created and are no longer maintainted. Roughly 30% of OAuth 2.0 implementations were created and dropped before the OAuth 2.0 specifications were formally done, so their completeness is dubious at best. Also, many of the smaller implementations done by hobbyists are only compatible with a single authentication provider, like Google, Facebook, or Twitter. This makes general use and testing difficult.

3.2.1 *Differences.* With so many implementations, and with a standard that is often vague, differences will arise. Some of the more important ones are:

—URL format - some use simple url arguments, some store data in JSON format.

—OAuth flow types - OAuth has 4 different authorization grant flows, and not all client implement them all. The most common one by far is the authorization code flow, which simply gets a token from the server to present later when requesting a resource.

—Storage options - OAuth leaves the question of how to store client secrets and other sensitive information up to the implementer and developers. The most common ones are MySQL and SQLite.

—Permission tests - for a legitimate user who requests permissions they do not have authorization for, it is often left up to the developer whether to throw and error and/or return only the applicable permissions.

## 3.3 OAuth Vulnerabilities

A lot of OAuth implementations, by themselves, do not offer security features - most of the time, these are left up to the user to implement, or to choose to use in their program. For that reason, it is sometimes hard to say definitively if an implementation is vulnerable to a certain attack or not. Here are the four most common issues seen in OAuth implementations:

(1) Auth Code Replay Attacks - when a provider sends an authorization code, they must delete it or expire it after it has been redeemed by the client (or after a designated short amount of time) to prevent an attacker from redeeming it themselves. The attacker could have obtained it via logs, hacked databases, or possibly from network traffic, and they could then use it to get an access token with access to the victim's account. None of the implementations examined provided any utilities to help manage this, and several had access logs where the auth codes were recorded as part of the URL requested.

(2) Redirect URI path traversal - if subdirectories are allowed to be set during URI validation, using tricks like "%2e%2e" or "/../" within the url can allow an attackers to traverse the server folder tree. The solution is to make sure the redirect URI is identical to the one registered, but this almost always left up to the library user as a choice, since there could potentially be applications that need it the other way.

(3) Access Token Abuse - mobile and server applications (ones with less human interaction) often use a supplied, pre-generated access token. If a single access token is supplied by the user for multiple application authorizations, a malicious app can take a user's access token and use it to gain access to the victim's account on the other applications. The solution to this is that the OAuth provider needs to check that the token was issued for the specific client ID that is requesting to use it. This is not something included in most OAuth server implementations, but high-profile providers like Facebook and Google do provide methods to check this.

(4) CSRF - OAuth provider endpoints must use CSRF tokens in their forms, or they become vulnerable to CSRF attacks, which allows client account hijacking. Most implementations are not involved in building the forms, but this vulnerability occured in a few that were made for a specific content platform, like Django or Flask.

## 3.4 OAuth Provider Security

Using a single account for everything creates a single point of failure - if the password for that account is stolen, attackers could access your accounts across many different sites. This is one of the reasons password re-use is frowned upon. Users need to be careful what companies they place this trust in. Ideally, passwords should be stored securely, at least salted and hashed, to make them difficult to crack, but given enough time and resources even that can be broken.

There have been countless instances of password hashes being stolen recent years - companies are constantly under attack. Companies that are OAuth providers are no exception, and there are many recent examples. In 2013, GitHub suffered a massive brute force attack which compromised many user accounts. In response, the company had to improve its rate limiting and crack down on weak passwords [19]. The next year, Yahoo! had the usernames and passwords of email accounts stolen [21]. In November of 2015, Amazon sent out password reset notices in response to a suspected security issue with the way passwords were transmitted from or stored on user devices [18]. Around the same time, Instagram had an issue with its OAuth API that allowed third-party apps using its services to steal and store user passwords [20]. In 2016, Tumblr announced that it only just now learned of a breach that occurred almost 3 years ago (in 2013) [16], where attackers stole the database of salted and hashed passwords. As a result, all affected accounts were required to change their passwords. Tumblr is considered one of the top an OAuth providers. Another one of the top providers, LinkedIn, suffered a massive breach of a similar nature in 2012, which affected over 117 million accounts [17]. However, it wasn't until 2016, nearly four years later, that they discovered the full extent of the breach and forced most of those accounts to reset their passwords.

Although companies that provide OAuth authentication may have the best of security intentions, the fact is that most companies do not immediately know when they have been hacked. A 2013 survey done by Verizon [15] found that 70% of company breaches were discovered by someone outside the company. Also, as demonstrated by recent examples, breaches (which tend to happen in hours or over a few days) are often not discovered until months or years later. The same survey found that the majority of breaches (66%) took months to discover, and 4% took years. During all that time, the attackers are free to make use of the data that they've stolen, without consumers even realizing that their account has been compromised. And, once the breach is discovered, containment usually takes days (41%) to months (22%). So if someone puts their all of their trust in a single company, they need to be extremely sure that company has good security, which is difficult to do.

## 3.5 OAuth Privacy

Very few users realize that by using an app with an OAuth provider, they are often sharing data with that provider. The privacy policies of nine of the largest top companies that are OAuth providers were examined: Google, Facebook, Twitter, Dropbox, GitHub, Windows Live, SoundCloud, Box, and Foursquare. Two out of the nine companies (GitHub and Windows Live) do not have specific privacy policies in place.

Where policies existed, they had all been updated within the last 2 years, showing how the rapidly changing environment of data collection affects service providers. API providers were split on change notification policies. Facebook, Dropbox, Box, and Foursquare all promise to notify developers via email when the policies change substantially. Box has the strongest guarantee, as they explicitly promise a notice 30 days in advance of the changes. On the other hand, Google, Twitter, and SoundCloud make no promises about change notifications and expect interested parties to simply check their webpage often.

Only Foursquare made it explicit that developers must say that their app shares information with the company, although Facebook, Twitter, Dropbox, SoundCloud, and Foursquare all require applications using their APIs to have users agree to privacy policies. A sample of a handful of mobile apps using the Facebook API showed that none of them included information about Facebook's data collection in their privacy policies (only that for the app itself). So these things are most likely not being enforced.

Google, to perhaps no ones surprise, collects the most information and reserves the most rights for itself. It gathers every piece of data imaginable except for strictly legally regulated things, like health. It even goes out of its way to specify that app developers are not allowed to try and anonymize anything or try and prevent Google's heavy monitoring. Google even creepily says it is trying to figure out "the people who matter to you most". Google and Facebook also go one step further than other providers and give themselves the right to access the developer's app or website content for any purpose as long as they are using the providers API. The information is not limited to the provider either - Google, Facebook, Twitter, and Foursquare all reserve the right to send the information they gather to "interested" third parties.

3.5.1 *Tracking.* Because of its flexible, URL-based data format, OAuth integrates with mobile applications much more easily than traditional SSO methods. As a result, it is hugely popular in mobile apps. Previously, websites like Google and Facebook tracked users through cookies in the browser, but this did not expand well to mobile. OAuth offers the capability to bridge that gap. Because applications must register with providers, the provider can tell exactly which apps are being used with a particular account, and what data is requested. OAuth also allows providers to specify extra fields in the data, meaning any extra data could be sent along with the authorization. All companies except Foursqure have clauses that allow them use data gathered from the OAuth APIs in tracking and ad targeting for the end user. So not only can they track you - they already are, and they have been working on it long enough to integrate it into publicly-available legal documents.

## 3.6 Other SSO Methods

Some other SSO protocols are described below in comparison with OAuth.

3.6.1 *SAML*

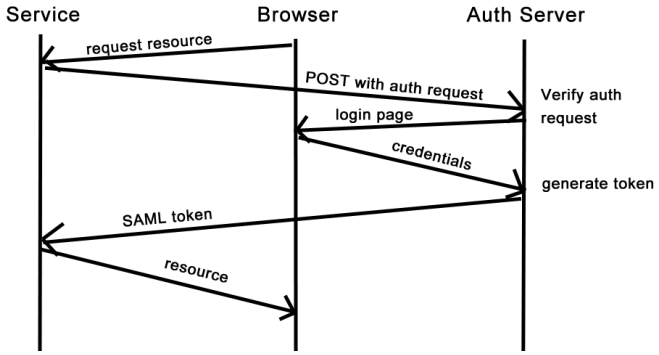


Fig. 2: SAML flow

SAML is the most commonly used SSO solution, and it provides both authentication and identity management. Like OAuth, it has many different kinds of flows that can be used. The most common scenario is pictured above. It has a much stricter data format than OAuth, but it is faster because it avoid the extra network round trip that OAuth takes to the provider. On average, for a very minimal setup, SAML was 11.29% faster than OAuth, a difference of roughly 50 milliseconds per request.

| **OAuth 2.0 (2012)** | **SAML 2.0 (2005)** |
|---|---|
| - Vague/young standard | + More mature standard/tools |
| - Does not contain user identity | + Signing/user identity included |
| - No authentication (but can be added) | + Does authentication and identity management |
| - Requires an extra round trip for packets | + Fewer packets sent in general |
| - Slower | + Faster |
| + Flexible token-based authorization | - Stricter XML format |
| + Could even use SAML's XML format | - Sends more data over the wire in general |
| + Easy to invalidate access tokens | - A bit more difficult to revoke access |
| + Mobile works with no mods | - Difficult to use with mobile |

SAML is difficult to use on mobile because of the way it uses the HTTP POST body. Usually mobile apps have no access to this. It makes redirecting a login request to a mobile app extremely counter-intuitive. There are ways in can be done, and has been done successfully, but OAuths simple URL parameter passing make it much easier to use in mobile environments.

3.6.2 *CAS*

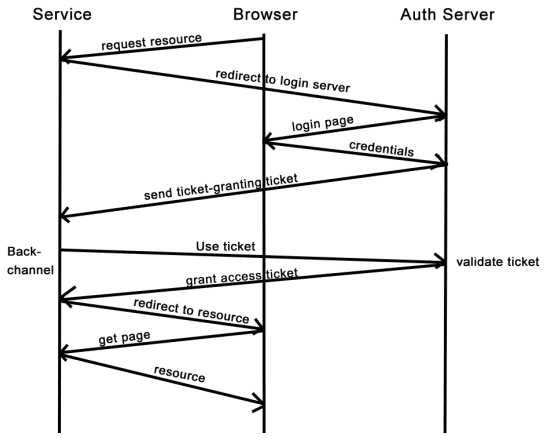


Fig. 3: CAS protocol

The CAS protocol (not to be confused with the CAS server software suite by the same authors) provides both authentication and authorization. However, it requires an extra network round-trip and sends a lot more data than OAuth, so overall it runs more slowly. It also has a less flexible XML data format. It works for mobile devices if an extra REST API is added.

In terms of maturity, there is no real contest - CAS has an enterprise-grade implementation under active development with full software support. Its specification is detailed and much more complete than OAuths. However, it really only has this one implementation (whereas OAuth has dozens), and because it is enterprise, its a huge software suite that takes a lot more effort to install than any OAuth implementation. Small-scale apps and even small business are probably going to lean towards OAuth just because of that. On the other hand, CAS has a professional security response team, and a single implementation means that bugs don't linger in old implementations once they are fixed.

3.6.3 *Cosign*

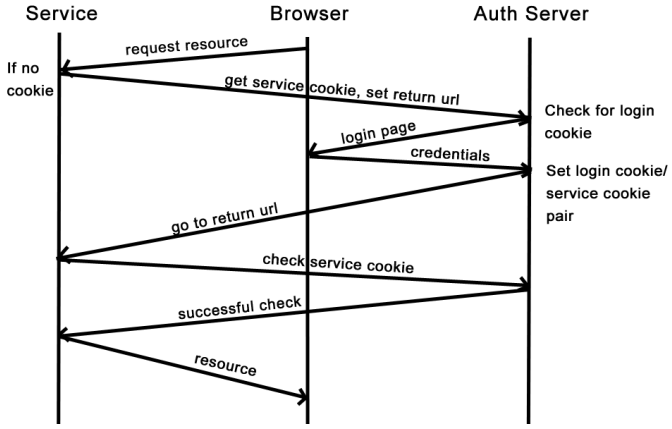


Fig. 4: CoSign protocol

Like OAuth, CoSign is more of a framework than a full-fledged protocol. They both work with many different kinds of authentication servers, and they both use URL query parameters to pass information between endpoints. They both avoid giving the service application user identity credentials by default. However, CoSign is clearly designed for user-based interaction (since it was designed as a university SSO solution), whereas OAuth is centered around applications. CoSign provides authentication, but not authorization, whereas OAuth is the opposite. Cosign uses a back-channel to the server to check a user's access, whereas OAuth makes an extra network round trip, which results in CoSign being slightly faster.

CoSign seems to suffer from a lack of clear security design, even more so than the vague OAuth standard. A few of the larger problems are:

—SSL is recommended but not forced, whereas OAuth requires it.

—HttpOnlycookieprotection is available,but not documented, and is turned off by default.

—When logging out, there is a delay where the user still has access to the service because of a cache timeout, and the service server must do extra work to mitigate this.

—It is open to CSRF attacks - CoSign does not check that the same user that was authenticated is the one requesting access. It only checks that the service cookie has a login cookie. This can be mitigated using IP checking or Kerberos tickets, but neither of these are on by default.

## 4. PASSWORD MANAGERS

Twenty of the most-recommended password managers were evaluated in this study: 1Password, Dashlane, Trend Micro Password Manager, IdentitySafe, KeePass, Keeper, Keychain Access (iCloud Keychain), LastPass, LogMeOnce, MSecure, NeedMyPassword, PassPack, Password Boss, True Key (formerly PasswordBox), Password Depot, Password Genie, Roboform, SplashID Safe, Sticky Password, and Zoho Vault. Additionally, the password managers of Firefox, Chrome, and Safari were evaluated for their autofill properties.

### 4.1 Cryptography

There are a lot of different pieces of cryptography that go into password manager vaults. First, the master password itself must be turned into a key. This is usually done with many iterations of the PBKDF2 key generation algorithm. The number of iterations ranges from 100,000 (1Password) to 1,000 (Keeper, Zoho Vault). In a secure password manager, neither the password nor the key is stored anywhere - instead the key is recomputed whenever the password is entered.

The exception to this key generation approach is PassPack. It, perhaps unwisely, uses its own custom key generation algorithm, citing parallelism issues. This key generation algorithm involves taking every other byte of a SHA-256 hash, then doing several rounds of salted hashing.

This master key is used to encrypt the account data that the user wants to store. 90% of the programs surveyed used AES-256 to encrypt the vault. The only real exception was MSecure, which uses Blowfish-256.

For a full list of encryption methods and their variations, see Appendix C.

### 4.2 Benefits

In addition to cutting down on the number of passwords to remember, password managers have added many beneficial new features.

75% of them offer two-factor authentication, and 60% have support for some form of biometrics, with fingerprint being the most common. Several of the higher-end programs also offer Active Directory/LDAP integration for enterprise users.

Besides simply storing your passwords, almost all password managers will also generate secure passwords for new accounts and give you a strength score for your existing passwords, so you can see which ones are weak and should be changed. The newest benefits also include password managers that can automatically change your password and those that notify you if you may be affected by a security breach at a company where you have an account (for instance, if their password hashes were stolen). Right now, the only password managers that do both are Dashlane and LastPass.

50% of password managers also offer keylogger protection in the form of virtual keyboards. Rather than typing, the user clicks on the keys of a displayed keyboard, whose input is translated by the password manager. This prevents a keylogger from picking up keystrokes from the main keyboard. For a full table of which password managers have which features, see Appendix B.

## 4.3 Browser Extension Autofill Vulnerabilities

Browser extensions are extremely convenient - they allow username and password fields to be filled with minimal or no user interaction. Unfortunately, this opens some avenues of attack for an active network attacker. For instance, if an attacker controlled a router in a cafe, they could serve a router login page with invisible forms, embedded iFrames, or pages that pop up behind the current window. When the victim goes to log in, their browser extension might automatically fill in these fields, and then that data could be sent back to the attacker with some Javascript.

As a continuation (and re-evaluation) of the research done by Silver et al [6], several different possible autofill scenarios were tested on the password managers with browser extensions:

(1) Default - a login page with a username and password field, served with valid HTTPS.
(2) HTTP - the same login page, but now served with HTTP instead of HTTPS
(3) Different form action - the form POSTs to a different url
(4) Autocomplete=”off” - this attribute is put in both the fields and the form header
(5) iFrame - the original login form, placed in an iFrame in a page with the same origin/domain.
(6) Modified field name - the name and id of the fields were changed
(7) display=none - this CSS attribute was applied to the form, making it invisible.
(8) opacity=0 - this CSS attribute was applied to the form, making it invisible.
(9) Pop-under - the original login page is opened in a background window.

Some password managers with browser extensions (like Identity Safe, Password Depot, Password Genie) were not evaluated because they were Windows-only, and a customizable Windows machine was not readily available. The browser password managers for Safari, Chrome, and Firefox were also evaluated, for a total of 14 different password managers. Safari and Chrome actually defeated the pop-under code (the window opened on top instead), so extensions were tested in Firefox for this scenario.

There were six different actions that could happen in each scenario:

—Auto - auto-fill both fields and auto-submit
—Fill - auto-fill both fields
—User - auto-fill only the username field
—Click - requires the user to click one button, then it fills and submits
—Click-Fill - requires the user to click one button to fill both fields, but does not submit the form
—None - no action taken or available from the password manager

Auto is the most dangerous behavior and also the most convenient. The user literally has to do nothing, but that also means an attacker requires no user interaction. Only Dashlane, LogMeOnce, and TrueKey used Auto - Dashlane and True Key used it for the default and pop-under scenarios, and LogMeOnce for every scenario except the ones that made the form invisible.

HTTP pages were, for the most part, distrusted by the password managers. LogMeOnce and Dashlane still both used Auto on the HTTP page, and StickyPassword and Firefox still used Fill. But every other program either required user interaction or refused to help with filling the fields at all. Since sending credentials in the clear is a very bad idea, it seems like the default should be to refuse to use HTTP. Surprisingly, no one gave any extra warning to the user about the page being insecure.

Almost no one recognized or cared when the form action was different. The action in this case was the same as the default except in the case of Dashlane, which used Fill instead of Auto (a minor downgrade). The modified field name only mattered to LastPass, but this may have been because it broke the way it saved the data (it refused to do anything in this scenario).

It was discovered that the autocomplete=”off” attribute is completely ignored by every program. It did not stop any auto-fills or click-fills. The autocomplete attribute is not particularly new, so it seems people have simply chosen to ignore it.

29% of these password managers recognized when the login page was in an iFrame: Dashlane, PassPack, SpashID Safe, and Sticky Password. These 4 refuse to do anything when the login page is in an iFrame. All other password managers continued their default behavior in this scenario. iFrames are particularly dangerous, because they can be inserted in any page, meaning an attacker could put them in any weak same origin page, or (if the site did not use some form of framebusting) on their own site to harvest credentials.

Most programs treated the two scenarios that made the forms invisible the same. The exception was Dashlane - in the display=none scenario, it only auto-filled the user field, whereas when opacity=0 it filled both. Keeper, LogMeOnce, SplashId Safe, Sticky Password, and Chrome all recognized both cases and treated them the same (usually by not filling anything). That means 57% of these password managers did not recognize when a form was being hidden, which is bad for users, because it means they could populate a form that the user isn’t even aware of.

In the case of the pop-under window, only four password managers treated it differently than a normal login window: 1Password, Keeper, PassPack, and SplashID Safe. Opening and closing a large number of pop-under windows could also be an advantage to an attacker looking to harvest credentials - it would be safer if only the window currently in focus was considered by the browser extensions.

For the full table of autofill behaviors, see Appendix F.

## 4.4 Web App Issues/Vulnerabilities

4.4.1 *Protecting the Master Password.* One of the most common security claims made on any password manager press release

is that your master password is never ”transmitted or stored anywhere”. And while this may be true in most cases, the way the statement is phrased is somewhat misleading. The master password may not be stored or transmitted, but what if the encryption key derived from it is? The key is just as useful (more so, in fact) than the password itself. And what if a derivative of the password is stored, such as a hash value for login? The claim is still true, but the security is now more questionable. For instance, in 2015, LastPass servers were breached, and a large amount of data was stolen, including user master password derivatives [3]. In their case, LastPass claims the derivatives were the master password run through over 100,000 iterations of PBKDF2, and thus extremely difficult to crack. However, it still recommended that users change their master password because of this.

Of the twenty password managers surveyed, 11 of them had web applications that accessed the saved passwords. Now, there are many ways a web application could be vulnerable, but some concerns are unique to password managers. A password manager web app requires the user to enter their master password into a web form, and this can be problematic if any information about the master password is stored on the server or passed over the network.

There are three different strategies that password managers have in their web applications. The first and simplest is that they treat the master password like an account password - the password or some derivative is sent to the server, who then checks it against a stored value and tells the client if the password was correct. This is how LastPass, 1Password, LogMeOnce, NeedMyPassword, and SplashID Safe work.

The second, slightly more complicated method uses two passwords: a traditional account password for logging in and a master password for the vault encryption. The user must enter both to see the data. The account password is treated like a normal account password and compared to its hash, while the vault password is used only for decryption on the client side. The password managers that use this strategy are Trend Micro, Zoho Vault, and PassPack.

The third and final method is probably the most secure: no password data is sent over the network at all. The master password is used to generate a key, and that key is used to attempt decryption on a known value. If the decryption is successful, the user is logged in, otherwise, the application knows the password is wrong. Dashlane, Keeper, and Roboform all use this method.

In this study, 5 other password managers (in addition to LastPass) were found that could be vulnerable to a breach of the kind suffered by LastPass: 1Password, LogMeOnce, NeedMyPassword, PassPack, and SplashId Safe. These all send some derivative of the password or master key over the network to the server. NeedMyPassword and SplashID Safe both sent the password over the network in its original form. PassPack and LogMeOnce both send the derived master key over the network. 1Password sends a salted PBKDF2 hash of the password, similar to LastPass.

For full details on the operation and what is sent over the network for different password managers, see Appendix E.

4.4.2 *Protecting the Data.* The data in a password manager vault should, ideally, be accessible to no one but the user. Secure web application accomplish this by doing all encryption and decryption on the client side in the browser, so that the server (and any network attacker) never see the data in the clear. Almost all the programs use this approach - however, there are two exceptions. NeedMyPassword and SplashID Safe both send the vault data unencrypted (within a TLS connection) to the client. This means that either they are decrypted server-side, or they are not encrypted at all, which means that the company that owns the password manager has access to the data. If that company were to be hacked, the hackers could see the data as well, either by stealing or by planting malware on the server to send the decrypted data to them. I suspect that NeedMyPassword does not encrypt the data at all - it claims to use ’256-bit’ encryption, but that seems to just be referring to its TLS key size. SplashID Safe claims to use AES-256, so it is more likely there that the data is being decrypted server-side. Either way, both practices are insecure.

## 4.5 General Web Security

The 11 password managers with web apps were also evaluated for general web security practices. Every password manager on the list used HTTPS/TLS, but not all were equally secure. LogMeOnce and LastPass used HTTPS, but not on all elements of the page - some images and scripts were loaded over HTTP. This is insecure because an active network attacker could intercept and modify those elements with malicious content, possibly content that could steal any usernames and passwords entered on the page. 8 of the 11 password managers made use of the Strict-Transport-Security HTML header, which says that the page can only be accessed with HTTPS.

Almost all pages used the same default TLS settings: TLS version 1.2 with a 128-bit key and the ECDHE, RSA, AES-128, GCM, SHA-256 cipher suite. There were two exceptions to this. NeedMyPassword uses TLS version 1.0, with a 256-bit key with RSA, AES, and SHA-1. Trend Micro uses a 112-bit key with RSA, 3DES, and SHA-1, a surprisingly old combination.

There are also the typical web security considerations: cookie protection, CSRF attacks, and XSS attacks. Six of the 11 programs used HTTPOnly cookie protection (meaning the cookies can only be accessed during a web connection, and not a client-side script), and 2 apps (Keeper and NeedMyPassword) did not store any cookies. HTTPOnly is good protection against malware or malicious browser extensions. Surprisingly, only one program (SplashID Safe) made use of CSRF tokens in their forms. Most programs had no Content Security Policy header to mitigate XSS attacks. LogMeOnce had this header, but it had extremely broad permissions, including an ’unsafe-eval’ exemption to run untrusted Javascript. Only 3 password managers used the X-XSS-Protection HTML header: LogMeOnce, Keeper, and Zoho Vault.

For a table of all the measures used, see Appendix D.

4.5.1 *iFrame Embedding.* iFrames are of particular concern because of the potential for tricking browser extensions into autofilling and for phishing a user with part of a legitimate site in an iFrame in a malicious site. So it is beneficial to password managers to ensure their login page cannot be contained in an iFrame (this is called framebusting). 5 of the 11 programs did not make use of the X-Frame-Options HTML header for protection: 1Password, LastPass, LogMeOnce, NeedMyPassword, and Roboform.

## 4.6 Browser Password Managers

Browser-based password managers are generally not recommended, and there are several reasons why.

(1) Encryption schemas for these tend to be weaker (Chrome and IE use 3DES) than other password managers.

(2) Browser extensions can access them. A single malicious extension could steal all your passwords.

(3) They are usually encrypted under the users account in the system keychain. So any malware running at the user level can

read them, or anyone who sits down at the computer while the user is logged in.

Firefox is the only browser that doesnt encrypt its logins in the user account. Instead, it has an optional master password that is turned off by default - if no master password is set, a blank password is used.

### 4.7 Recommended Password Manager

The password manager market is very saturated - theres a huge number of programs with an equally large variety of offerings. So there should be no reason to use a substandard program. Of the 20 password managers examined, I would recommend Dashlane.

Dashlane is relatively new the market, but it has a well-rounded suite of features, including the coveted breach alerts and automatic password changes. It works on all major platforms and has an easy yet slick-looking user interface. It is free for most users, and offers cloud storage for a price.

From the security perspective, Dashlane is one of only 3 password managers that lay out all their security practices in a security whitepaper [4], so they are not relying on security through obscurity. They use decent encryption, with the second-highest number of PBKDF2 rounds, and they do not pass any password-related information over the network. However, it is recommended that you turn off the auto-login feature in order to make it harder for active network attackers to exploit the auto-fill feature.

## 5. SSO VS. PASSWORD MANAGERS

### 5.1 Security

Both SSO and password managers protect against a passive network attacker by using TLS (OAuth requires the use of TLS), assuming the attacker cannot crack the TLS after the fact in time for it to be useful.

For an active network attacker, it is more variable. Some password managers have insecure practices that allow a man-in-the-middle attack to steal the password or key and the encrypted data. However, those that pass no password information over the network are immune to this particular approach. But active network attackers can still exploit autofill vulnerabilities for many password managers. The vulnerability of a password manager can be greatly reduced by simply not using a browser extension, but sometimes that is not an option. With SSO, it depends on the security of the provider - Google probably has decent security, but not every provider may, as seen in the password breach survey. In general, SSO protects against active network attackers better than password managers - OAuth's security considerations were specifically designed against an active network attacker[13].

SSO offers no protection against a malware attacker - the user enters their password as usual and it can be stolen in a variety of ways with malware on the machine. And since in this case the user is using this password for many applications, it is more valuable to attacks and more detrimental if it is stolen. Password managers can offer protection against keyloggers and clipboard sniffers. While this isn't a perfect solution (once malware is on your machine, there's no knowing what it will do), it is at least partial protection, which SSO does not offer. Using two-factor authentication with either method offers more protection, because a stolen password does not automatically compromise your account.

Against a physical attacker, the results are also mixed. If you leave your computer unlocked, or if your account password is bypassed, an attacker can use passwords stored in Chrome, IE, or the iCloud Keychain. Most password managers do not have this issue - they lock the vault when it is not in use. However, if the user leaves the computer open AND the vault unlocked, a physical attacker can read all the passwords even without knowing the master password. With OAuth, a physical attacker does not gain any advantage over a network attacker unless the user stores their SSO password somewhere insecure (like on a post-it note or in a plaintext file). So if the user is smart, they will not have any extra vulnerability, but at the same time SSO does not inherently offer any protection against physical attackers.

### 5.2 Privacy

Password managers definitely have the advantage over OAuth where privacy is concerned, because it puts the user in control of their data. Cloud accounts for password managers are a risk that puts your data in the hands of someone else, but if you verify that all decryption and encryption happens client-side, it might be a trade-off worth taking. On the other hand, OAuth puts all of the knowledge on the provider's side - they can track when you login, from what IP addresses, and what apps you use and what data they request. Providers can even track you across mobile devices, which previous cookie-based tracking could not accomplish.

### 5.3 Ease of Use

For the user, OAuth is easier to use because they don't have to change anything. They just use the accounts they already have with Facebook or Google, with the same amount of logins. Using a password manager requires users to install something and change their routine. However, this difference is fairly small - most password managers are easy to use, and this would not be sticking point for most users.

### 5.4 Cost

The cost for each method is placed on different people. In OAuth, the additional burden of cost is placed on the authentication provider and application authors. They have to program OAuth and invest in server capacity to meet the demand of the user, and it doesn't cost the user a dime (except in ads). But for password managers, the cost falls to the user. Most password managers have a free version, so the user doesn't necessarily have to pay anything. But for cloud storage or some extra features, many password managers have a yearly fee. 1Password is the most expensive, at $49.99, but many are less than $10.

## 6. RECOMMENDATIONS

### 6.1 For a Home User

The best solution for a home user is to pick a good password manager (like Dashlane) and use it regularly. This puts all the control in the user's hands, and generally results in more secure passwords being used. Password managers are free (or cheap), so anyone can use them. OAuth enables too much tracking behavior to be worth it for a privacy-conscious user.

### 6.2 For a Small Startup

The initial cost of setting up a SSO solution is high, so for a small startup, it would be cheaper and easier to invest in an enterprise-level password manager. While these aren't free, they are still relatively cheap, and offer professional support, Active Directory integration, team accounts and more. However, since most password

managers are on a yearly or monthly subscription plan for businesses, the long-term cost might eventually outstrip that of a SSO solution, so it depends on the growth rate of the company.

### 6.3 For a Large Corporation

A large corporation with a lot of internal applications should probably set up their own SSO authentication. Although it is a steep curve up front, it is a relatively low-maintenance solution without recurring costs. It also makes life much easier for employees to be able to sign into everything at once. If the corporation requires employees to hold accounts in outside applications, password managers may also be a good additional investment.

## 7. CONCLUSION

There is currently no 100% satisfactory solution to do the password problem, but we are making progress. The large market for both authentication implementations and password managers has ensured that users at least have a choice, and they can take their privacy and security into their own hands. The problem and its solutions are both complicated - a lot of time was spent puzzling out the inner workings of protocols and cryptographic schemes just to understand what is going on. The main lesson to be learned here is to take nothing at face value - think critically about the claims made by vendors. Password managers are not secure just because their website claims they have "military-grade encryption" any more than Facebook has your best interests at heart when it pushes its login services all over the internet.

## REFERENCES


Research Now *CONSUMER SURVEY: PASSWORD HABITS*, (CSID) (2012)
https://www.csid.com/wp-content/uploads/2012/09/CS_PasswordSurvey_FullReport_FINAL.pdf

Matteo DellAmico et al. *Password Strength: An Empirical Analysis*, (InfoCom) (2010)
http://www.eurecom.fr/en/publication/2910/download/rs-publi-2910_1.pdf

Joe Siegrist *LastPass Security Notice*, (LastPass) (2015)
https://blog.lastpass.com/2015/06/lastpass-security-notice.html/

*Dashlane Security Whitepaper*, (Dashlane) (2011)
https://www.dashlane.com/download/Security-Whitepaper-Final-Nov-2011.pdf

DLi, Zhiwei et al. *The Emperors New Password Manager: Security Analysis of Web-based Password Managers*, (Usenix Security) (2014)
http://devd.me/papers/pwdmgr-usenix14.pdf

D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson. *Password Managers: Attacks and Defenses*, (Usenix Security) (2014)
https://crypto.stanford.edu/~dabo/pubs/abstracts/pwdmgrBrowser.html

P. Gasti and K. B. Rasmussen. *On The Security of Password Manager Database Formats*, (University of California) (2012)
https://www.cs.ox.ac.uk/files/6487/pwvault.pdf

R. Chatterjee, J. Bonneau, A. Juels, T. Ristenpart. *Cracking-Resistant Password Vaults using Natural Language Encoders*, (University of California) (2012)
http://www.jbonneau.com/doc/CBJR15-IEEESP-cracking_resistant_password_vaults.pdf

Sun, S., Beznosov, K. *The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems*, (CCS) (2012)
http://css.csail.mit.edu/6.858/2013/readings/oauth-sso.pdf

J. Bonneau, C.Herley, P. van Oorschot, F. Stajano *The quest to replace passwords: a framework for comparative evaluation of Web authentication schemes*, (University of Cambridge) (2012)
http://css.csail.mit.edu/6.858/2013/readings/oauth-sso.pdf

Andrey Labunets *OAuth 2.0 and the Road to XSS*, (Hack In The Box Security Conference) (2012)
http://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Andrey%20Labunets%20and%20Egor%20Homakov%20-%20OAuth%202.0%20and%20the%20Road%20to%20XSS.pdf

P.Hu and W. C.Lau *OAuth App Impersonation Attack*, (BlackHat) (2014)
https://www.blackhat.com/docs/us-14/materials/us-14-Hu-How-To-Leak-A100-Million-Node-Social-Graph-In-Just-One-Week.pdf

T. Lodderstedt,M. McGloin, P. Hunt *OAuth 2.0 Threat Model and Security Considerations*, (IETF) (2013)
https://tools.ietf.org/html/rfc6819#section-2.2

*List of OAuth Providers*, (Wikipedia) (2015)
https://en.wikipedia.org/wiki/List_of_OAuth_providers

Verizon RISK Team *2013 Data Breach Investigations Report*, (Verizon) (2013)

*Third Party Breach*, (Tumblr Staff) (2016)
https://staff.tumblr.com/post/144263069415/we-recently-learned-that-a-third-party-had

Brian Krebs *As Scope of 2012 Breach Expands, LinkedIn to Again Reset Passwords for Some Users*, (Brian Krebs) (2016)
http://krebsonsecurity.com/2016/05/as-scope-of-2012-breach-expands-linkedin-to-again-

*Amazon Forces Password Resets after Possible Security Breach*, (Security Week) (2015)
http://www.securityweek.com/amazon-forces-password-resets-after-possible-security-bre

Adi Robertson *Weak GitHub passwords lead to account security breach*, (The Verge) (2013)
http://www.theverge.com/2013/11/20/5126906/weak-github-passwords-lead-to-account-security-breach

Selena Larson *Instagram restricts API following password breach, will review all apps going forward*, (The Daily Dot) (2015)
http://www.dailydot.com/technology/instagram-api-restrictions/

*Yahoo email account passwords stolen*, (Yahoo!) (2014)
https://www.yahoo.com/news/yahoo-email-account-passwords-stolen-002044026--financ
html?ref=gs

# APPENDIX

## A. PASSWORD MANAGER PLATFORMS

Table I. : Password Manager Platforms

| | Desktop | Mobile | Browser Extension | Device Sync | Web Access | Provider Auto Backups | 3rd Party Cloud Backups |
|---|---|---|---|---|---|---|---|
| **1Password** | W,M, L-W | All | G,F,S,O | Yes | Yes | Yes | Yes |
| **Dashlane** | W,M | A,I | All | Yes | Yes | No | No |
| **Trend Micro Password Manager** | W,M | All | G,F,IE | Yes | Yes | Yes | No |
| **Identity Safe** | W | A,I | G,F,S,IE | Yes | No | No | No |
| **KeePass** | All | All | All | No | No | No | Yes |
| **Keeper** | All | All | G,F,S,IE | Yes | Yes | Yes | No |
| **Keychain Access** | M | I | S | Yes | No | Yes | No |
| **LastPass** | All | All | All | Yes | Yes | Yes | No |
| **LogMeOnce** | W,M | A,I | All | Yes | Yes | Yes | Yes |
| **MSecure** | W,M | All | In dev | Yes | No | No | Yes |
| **NeedMy Password** | None | None | None | No | Yes | Yes | No |
| **PassPack** | In dev | None | G,F,S,IE | Yes | Yes | No | No |
| **Password Boss** | W | All | G,F,IE | Yes | No | Yes | No |
| **True Key** | W,M | A,I | G,F,IE | Yes | No | No | No |
| **Password Depot** | W | All | G,F,IE,O | No | No | No | Yes |
| **Password Genie** | W,M | A,I | G,F,S,IE | Yes | No | Yes | No |
| **Roboform** | All | All | All | Yes | Yes | No | No |
| **SplashID Safe** | W,M | All | G | Yes | Yes | Yes | No |
| **Sticky Password** | W,M | A,I | All | Yes | Yes | Yes | Yes |
| **Zoho Vault** | None | A,I | None | Yes | Yes | Yes | No |

Table II. : Key

| **Operating Systems** | | | **Browsers** | |
|---|---|---|---|---|
| A = Android | L-W = Linux with Wine | W = Windows | G = Google Chrome | S = Safari |
| I = iOS | In dev = in development | M = Mac OS X | F = Firefox | O = Opera |
| Wp = Windows Phone | | L = Linux | IE = Internet Explorer | |

## B. PASSWORD MANAGER FEATURES

Table III. : Security Features

| | **Secure Password Generator** | **Password Strength Metrics** | **Automatic Password Changes** | **Security Breach Alerts** | **Keylogger Protection** | **Biometrics** | **Multi-Factor Auth** | **AD/ LDAP** | **Pro Support** |
|---|---|---|---|---|---|---|---|---|---|
| **1Password** | Yes | Yes | No | Yes | Yes | fingerprint | Yes | No | Yes |
| **Dashlane** | Yes | Yes | Yes | Yes | No | fingerprint | Yes | No | Yes |
| **Trend Micro Password Manager** | Yes | Yes | No | No | Yes | No | No | No | No |
| **Identity Safe** | Yes | Yes | No | No | Yes | No | No | No | Yes |
| **KeePass** | Yes | Yes | No | No | No | No | Yes | No | No |
| **Keeper** | Yes | Yes | No | No | No | fingerprint | Yes | Yes | Yes |
| **Keychain Access** | Yes | Yes | No | No | No | fingerprint | Yes | Yes | Yes |
| **LastPass** | Yes | Yes | Yes | Yes | Yes | fingerprint | Yes | Yes | Yes |
| **LogMeOnce** | Yes | Yes | Yes | No | Yes | face, fingerprint | Yes | Yes | Yes |
| **MSecure** | Yes | Yes | No | No | No | fingerprint | No | No | No |
| **NeedMy Password** | No | No | No | No | Yes | No | No | No | No |
| **PassPack** | Yes | Yes | No | No | Yes | No | Yes | No | No |
| **Password Boss** | Yes | Yes | No | No | No | fingerprint | Yes | No | Yes |
| **True Key** | Yes | No | No | No | No | face, fingerprint | Yes | No | Yes |
| **Password Depot** | Yes | Yes | No | No | Yes | No | Yes | Yes | Yes |
| **Password Genie** | Yes | Yes | No | No | No | No | No | No | No |
| **Roboform** | Yes | No | No | No | No | No | Yes | No | Yes |
| **SplashID Safe** | Yes | Yes | No | No | Yes | fingerprint | Yes | Yes | Yes |
| **Sticky Password** | Yes | Yes | No | No | Yes | fingerprint | Yes | No | Yes |
| **Zoho Vault** | Yes | No | No | No | No | fingerprint | Yes | Yes | Yes |

## C. PASSWORD MANAGER CRYPTOGRAPHY

UNK = unknown - this means either the algorithm was performed server side or could not be viewed for other reasons (ie. cost).

Table IV. : Password Manager Cryptographic Algorithms

| | Keygen Algorithm | Salt (bytes) | Iterations | Hash | Database Encryption | Key Size (bits) |
|---|---|---|---|---|---|---|
| **1Password** | PBKDF2 | 16 | 100,000 | SHA-512 | AES | 256 |
| **Dashlane** | PBKDF2 | 32 | 10,000 | SHA-1 | AES | 256 |
| **Trend Micro Password Manager** | PBKDF2 | 16 | 2,000 | SHA-256 | AES | 256 |
| **Identity Safe** | PBKDF2 | UNK | UNK | UNK | AES | 256 |
| **KeePass** | PBKDF2 | 16 | 6,000 | SHA-256 | AES | 256 |
| **Keeper** | PBKDF2 | 16 | 1,000 | SHA-256 | AES | 256 |
| **Keychain Access** | PBKDF2 | UNK | 10,000 | UNK | AES | 256 |
| **LastPass** | PBKDF2 | 32 | 5,000 | SHA-256 | AES | 256 |
| **LogMeOnce** | PBKDF2 | 32 | 10,000 | SHA-1 | AES | 256 |
| **MSecure** | PBKDF2 | UNK | UNK | SHA-256 | Blowfish | 256 |
| **NeedMy Password** | UNK | UNK | UNK | UNK | UNK | 256 |
| **PassPack** | Custom | UNK | UNK | SHA-256 | AES | 256 |
| **Password Boss** | PBKDF2 | UNK | UNK | UNK | AES | 256 |
| **True Key** | PBKDF2 | UNK | UNK | SHA-512 | AES | 256 |
| **Password Depot** | PBKDF2 | UNK | UNK | UNK | AES | 256 |
| **Password Genie** | PBKDF2 | UNK | UNK | UNK | AES | 256 |
| **Roboform** | PBKDF2 | 16 | 1,000 | SHA-256 | AES | 256 |
| **SplashID Safe** | UNK | UNK | UNK | UNK | AES | 256 |
| **Sticky Password** | PBKDF2 | 16 | 3,000 | SHA-256 | AES | 256 |
| **Zoho Vault** | PBKDF2 | 16 | 1,000 | SHA-256 | AES | 256 |

## D. PASSWORD MANAGER WEB APP PROTECTIONS

A summary of some of the measures taken (or not taken).

Table V. : Protections used by web app password managers

| | HTTPS | TLS Version | TLS Key Size (bits) | TLS Cipher (default) | HTTP ONLY | Strict-Transport-Security | X-Frame Options | CSRF Token | X-XSS-Protection | Content-Security-Policy |
|---|---|---|---|---|---|---|---|---|---|---|
| **1Password** | Yes | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | Yes | No | None | No | No | None |
| **Dashlane** | Yes | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | No | Yes | Same origin | No | No | None |
| **Trend Micro Password Manager** | Yes | 1.2 | 112 | RSA, 3DES, EDE, CBC, SHA | Yes | Yes | Deny | No | No | None |
| **Keeper** | Yes | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | No | Yes | Same origin | No | Yes | Itself, Google, AWS |
| **LastPass** | Partial | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | Yes | No | None | No | No | None |
| **LogMeOnce** | Partial | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | Yes | Yes | None | No | Yes | Very long, adverts, 'unsafe-eval' |
| **NeedMy Password** | Yes | 1.0 | 256 | RSA, AES-256, CBC, SHA | No | No | None | No | No | None |
| **PassPack** | Yes | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | No | Yes | Same origin | No | No | None |
| **Roboform** | Login only | 1.2 | 128 | RSA, AES-128, CBC, SHA | Yes | Yes | None | No | No | None |
| **SplashID Safe** | Yes | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | Yes | Yes | Deny | Yes | No | None |
| **Zoho Vault** | Yes | 1.2 | 128 | ECDHE, RSA, AES-128, GCM, SHA-256 | No | Yes | Same origin | No | Yes | Non |

## E. PASSWORD MANAGER WEB APP METHODS

This section goes into greater detail about the operation and network traffic of password managers with a web interface. In all cases, these things happen within a TLS session, so sending something in ”plaintext” means it has no addition encryption other than TLS.

(1) 1Password - A group generator (g) is raised to the modulus power of a salted password hash. This value is sent to the server for comparison to check the password validity, so the server must store at least this value when the account is created. If the password is correct, the key is generated using Javascript in the browser with 100,000 iterations of PBKDF2. The server sends the encrypted data, and the decryption is done client-side with the generated key.

(2) Dashlane - Their login process has two steps - email selection and password entry. Once you select the email, the server preemptively sends the encrypted data associated with that email. When the master password is entered, the key is generated using 10,000 rounds of PBKDF2. That key is then tested for decryption. If the decryption fails, the password is wrong. If it succeeds, the data is decrypted and displayed. No password-related information is ever passed over the network, so it is unlikely that Dashlane servers store any password hashes.

(3) Trend Micro Password Manager - Trend Micro has a two-step login. You first log into your Trend Micro account, then you enter your master password. The account password is treated like any normal password, and sent in a hashed version to the server during login. Then, the server sends back (in a cookie), an encrypted key and initial vector (IV) that it encrypted and stored server-side during the last session with this account. The master key is generated from the master password using PBKDF2 with 2,000 rounds. The master key is used to decrypt the key and IV sent by the server (which become a sort of session key), and to encrypt a new key and IV for next time. The vault data is then sent over - the entire vault is encrypted with the session key that was decrypted by the master key. Sensitive data (usernames and password) are also encrypted with the master key. The session key decrypts the vault on the client side to display the main page, and sensitive fields are decrypted on the client side as they are accessed.

(4) LastPass - When the password is entered, the master key is generated with PBKDF2 with 5,000 rounds. The key is then run through one more round of PBKDF2 and this value is sent to server to confirm the login, so the server stores this value. The username is passed over TLS in both plaintext and encrypted with the master key. Encrypted vault data is sent after login and decrypted with the master key on the client side. New entries are encrypted before being sent back to the server.

(5) LogMeOnce - When the password is entered, the master key is generated using 10,000 iterations of PBKDF2, with the username as a salt. This master key is then sent in plaintext to the server to confirm the login, so it possible that the server stores the keys. The server sends over the encrypted data, and all decryption and encryption of new records is done client-side.

(6) NeedMyPassword - Everything, including the password, is sent in plaintext to the server. The data received from the server is also in plaintext. If there is any encryption/decryption beyond TLS, it all happens server side, so the server has access to your data and passwords.

(7) PassPack - Like Trend Micro, this has both an account password and an encryption password. They have a custom hash function: every other byte of the SHA-256 hash of the value, starting with the first byte. For the account login, the base64 value of the username is sent to the server, along with the custom hash of the accountpassword, and the account password appended to itself. For the encryption password, they send this data, plus the first two bytes of a double-SHA-256 hash of the password, plus the key and a custom salted hash, all in plaintext. They use a custom key generation algorithm, which is probably not wise: A double-SHA-256 hash of the password with a salt each time, where the salt is a pre-determined value appended to the password. The server determines if a login is valid by sending a message index value, so the server stores some if not all of the sent values for verification. The server then sends the encrypted data, and all decryption is done client-side.

(8) Roboform - On login, the base64 of the username and a 16-byte salt value are sent to the server, where the account is logged in. The server replies with the encrypted data, and decryption is done client-side. If the decryption fails, the password is judged to be incorrect.

(9) SplashID Safe - Password is passed in plaintext - the server tells the browser if it is correct or not, so it must store some password information to determine this. The data is similarly passed in plaintext, so any decryption happens server-side, meaning that the server has access to your data at some point.

(10) Zoho Vault - Like others, this has both an account password and an encryption password. The account password and username are transmitted in plaintext, so the server stores some password-related information for the account password. When the encryption password is entered, the server sends an encrypted JSON dict with one value: the timestamp of the last account login. The master key is generated from the encryption password using PBKDF2 with 1,000 iterations. The master key is used to decrypt the JSON - if it succeeds, the password is deemed correct, and the server sends the rest of the encrypted vault. Decryption happens client-side in the browser.

## F. PASSWORD MANAGER AUTOFILL PRACTICES

Table VI. : Autofill practices in password managers

| | **Default** | **HTTP page** | **Different Form Action** | **Autocomplete = off** | **iFrame** | **Modified Field Name** | **CSS display = None** | **CSS opacity = 0** | **Background Window** |
|---|---|---|---|---|---|---|---|---|---|
| **1Password** | Click | Click | Click | Click | Click | Click | Click | Click | None |
| **Dashlane** | Auto | Auto | Fill | Fill | None | Fill | User | Fill | Auto |
| **Trend Micro Password Manager** | Click | None | None | Click | Click | Click | Click | Click | Click |
| **Keeper** | Click | Click | Click | Click | Click | Click | None | None | None |
| **Keychain Access (Safari)** | Fill | Click-Fill | Fill | Fill | Fill | Fill | Fill | Fill | Fill |
| **LastPass** | Fill | Fill | Fill | Fill | Fill | None | Fill | Fill | Fill |
| **LogMeOnce** | Auto | Auto | Auto | Auto | Auto | Auto | None | None | Auto |
| **PassPack** | Click | None | Click | Click | None | Click | Click | Click | None |
| **True Key** | Auto | None | Fill | Fill | Fill | Fill | Fill | Fill | Auto |
| **Roboform** | Click | None | Click | Click | Click | Click | Click | Click | Click |
| **SplashID Safe** | Click-Fill | Click-Fill | Click-Fill | Click-Fill | None | Click-Fill | None | None | Click-Fill |
| **Sticky Password** | Fill | Fill | Fill | Fill | None | Fill | None | None | Fill |
| **Firefox** | Fill | Fill | Fill | Fill | Fill | Fill | Fill | Fill | Fill |
| **Chrome** | Fill | None | Fill | Fill | Fill | Fill | User | User | Fill |

Key:

—Auto - auto-fill both fields and auto-submit
—Fill - auto-fill both fields
—User - auto-fill only the username field
—Click - requires the user to click one button, then it fills and submits
—Click-Fill - requires the user to click one button to fill both fields, but does not submit the form
—None - no action taken or available from the password manager