∅pass: Zero-storage Password Management Based on Password Reminders

Giannis Tzagarakis
FORTH-ICS, Greece
gtzagarakis@gmail.com

Panagiotis Papadopoulos
FORTH-ICS, Greece
panpap@ics.forth.gr

Antonios A. Chariton
University of Crete, Greece
csd3235@csd.uoc.gr

Elias Athanasopoulos
University of Cyprus, Cyprus
athanasopoulos.elias@cs.ucy.ac.cy

Evangelos P. Markatos
FORTH-ICS, Greece
markatos@ics.forth.gr

ABSTRACT

A plethora of Internet services and applications require user authentication. Although many alternatives have been proposed, and despite the significant advancement in attackers' capabilities to perform password cracking, the most attractive authentication technology today, is still text-based passwords.

The last years, there is a rapid increase in the number of web services a user accesses in their everyday life. Most of these services (e.g., online shops, OSNs, chat clients, etc.) require their very own password, thus increasing the burden of password management on the user side. In this paper, we propose *∅pass*, a novel system that combines ideas from existing authentication methods, to offer a user-friendly mechanism to securely maintain accounts. *∅pass* works as a password manager, but it requires *zero storage* for the passwords: no password will ever get stored either in the user's device, or in a third-party database.

We implement *∅pass* as an extension for the popular Google Chrome browser, and we evaluate it by using the popular business-oriented social networking service LinkedIn. Early results from our performance tests show that *∅pass*, using a proactive strategy, can achieve more than 2 orders of magnitude better performance than the current state-of-the-art authentication mechanism.

CCS CONCEPTS

• **Security and privacy** → *Authentication*; **Access control**;

KEYWORDS

Password Management, Password Reminders, User Authentication

ACM Reference Format:

Giannis Tzagarakis, Panagiotis Papadopoulos, Antonios A. Chariton, Elias Athanasopoulos, and Evangelos P. Markatos. 2018. ∅pass: Zero-storage Password Management Based on Password Reminders. In *EuroSec'18: 11th European Workshop on Systems Security , April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3193111.3193113

EuroSec'18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5652-7/18/04.
https://doi.org/10.1145/3193111.3193113

1 INTRODUCTION

Text-based password authentication is based on the assumption that the users know something secret (memometrics). This authentication mechanism has been connected with many problems, such as leakage [13], and phishing [8] attacks. To remedy some of them, more sophisticated passwords are recommended. Yet, a UK study [18] presents that 70% of the users cannot remember complex or long passwords, and 44% of shoppers have abandoned at least one online shopping transaction because they were frustrated with the complexity of identity verification. 24% of these abandoned transactions were not taken elsewhere, as users canceled their online shopping attempt in general, resulting in *214 million pounds worth of net lost revenue for retailers.* Additionally, recent studies[20] have shown that users, on average, spend 12 full days of their lives searching for the correct username and password pair. If one tries to extrapolate this to the global online population, this results in a frustrating *productivity sink of 16.3 billion hours a year in total.*

Despite all of the above, still, text-based passwords remain the dominant authentication technique for web services today. And since they seem not to be replaced by anything, anytime soon, many alternatives have been proposed for hardening password authentication. One of these, Two-Factor Authentication (2FA), requires users to supply an additional password, most likely a token, for logging into the service. This token is received over a different, out-of-band, communication channel, such as cellural network, and then, users supply it along with their password over Internet. Recent studies measure 2FA adoption in probably the largest provider of this technology, Google, and found that less than 7% of its users have enabled 2FA [16, 21]. Other alternatives include Single-Sign-On (SSO) services, and password managers, which will be further discussed in Section 3.

In this paper, we combine some of the most important properties of the above alternatives, and deliver *∅pass*, a novel system for providing password-based authentication. The proposed system significantly raises the bar for the attackers, without degrading the users' experience. Using *∅pass*, passwords are stored neither at user's host, nor at a third party's cloud. When *∅pass* is to log a user into a web service, it triggers a Password Reminder process, resets the old password, and automatically creates a new session logging the user in. The same procedure will take place when the set cookie expires.

Compared to traditional authentication methods, there are a few important properties that make our approach resistant to modern password attacks. First of all, users' passwords are not stored in a single place and therefore cannot be leaked. Secondly, anytime a user wants to log into a service using ∅*pass*, a password-reminder process is taking place using an e-mail provider. The provider is invoked *only* in the login process, and not in further actions, as it happens with several SSOs, like Facebook Connect [14]. Thirdly, compared to traditional SSOs which generate authorization tokens, ∅*pass* does not. ∅*pass* allows a third party to *only vet* for the creation of a new user session. The third party, in contrast with SSOs, does not generate *any* access token that can grant access. Only the user's web browser has the credentials to authenticate with the target web site. This is important, because authorization tokens stored in third parties can be leaked exactly as it happens with passwords. Forth, ∅*pass* needs a single password per user. This password may be stored in a database, which can be leaked [13], however, the user can create a very hard password that can easily resist in cracking. Additionally, this (master) password can be protected by other means, such as 2FA.

Contributions. The contributions of this paper can be summarized in the following:

(1) We design ∅*pass*, a novel system which is based on existing ideas for authenticating a user anytime they want to log into a service. The authentication is achieved by triggering a new Password Reminder process, resetting the user's password. In addition, we propose two different proactive strategies to significantly improve the performance of ∅*pass*.

(2) To explore the feasibility of our approach, we implement a prototype of ∅*pass*, as a browser plug-in, for the Google Chrome browser.

(3) We evaluate ∅*pass* by using as a use case the popular business-oriented social networking service LinkedIn. Early results show that ∅*pass*, with a proactive strategy deployed, can perform at least 2 orders of magnitude better than one of the current state-of-the-art authentication mechanisms.

2 THREAT MODEL

In this paper, we assume attackers that can leak the databases of services where cryptographically hashed (and salted) passwords are stored. Attackers have the computational resources for cracking passwords based on dictionary words, but not for *reversing* state-of-the-art cryptographic hash functions. For instance, a long password, that is hashed with SHA256 can be cracked only if it contains easy-to-guess dictionary words.

In parallel, we assume attackers that can steal passwords using real-time phishing and, interactively, steal access tokens, serving as second factors, sent directly to the user. However, we do not assume powerful network attackers. For instance, a URL sent to the user's smartphone [9] cannot be captured by hijacking the user's network connection.

3 RELATED WORK

Current literature documents the needs of the current state-of-the-art approaches, which either replace, or facilitate, text-based password authentication [5]. Below, we briefly expand on each of these different fields.

3.1 Password Reminders

Every web service that supports password authentication, allows their users to reset their passwords, delegating thus trust to an e-mail provider. In case users forget their passwords, a password-reminder (PR) process can be initiated, using an e-mail account they have provided during their registration with the particular service. When such process takes place, users have to follow some steps (that vary depending on the web service), to reset their password. Since PR is critical to the operation of ∅*pass*, we describe here a generic description of the process, mainly inspired by the procedure carried out by a popular web service, LinkedIn.

3.2 Single-Sign On Services

There are several approaches trying to deal with the major disadvantage of traditional password authentication mechanism: the fact that the users have to memorize a separate password for each account they maintain. The technology of Single-Sign on (SSO), like OpenID [22], Facebook Connect [14, 17], the older, privacy-sensitive, Mozilla Persona [15], and Google [10], provide users with the ability of maintaining a single identity, for all the different applications, so they can authenticate themselves by using a single trusted identity provider, like Facebook or Google.

However, these providers may carry privacy-related risks, and also suffer from vulnerabilities themselves [27]. A recent study [25] associates the limited adoption of such services with several concerns regarding their relinquishing control of the user base, as part of outsourcing authentication. It is important to stress that SSO is a free-to-use technology, but their internal design may allow the SSO providers to track their users' actions in the web, reconstructing thus, a large part of their browsing history. Evidently, in the end, the users are called to buy better security, by paying with their privacy.

Moreover, the ability to use some SSO services depends on the location of the user. For example, Facebook Connect cannot be used in China, where Facebook is not accessible. That means websites need to support multiple SSO providers, if they want to be sure that any user can log in, but this still does not address the issue of people using one SSO and then travelling for a period of time in an area where this SSO is not available. Usually websites will also have the normal account system as a backup, which means that SSO may not trully solve the problem completely.

In ∅*pass*, we leverage the idea of a single identity, that is essentially responsible of all the user's authentication needs, and we couple this identity with an e-mail provider. ∅*pass*, in contrast with SSO, delegates trust to the e-mail provider, without revealing any privacy-sensitive information. Moreover, contrary to SSO, ∅*pass* is invoked only during the authentication process, which practically is rarely done [3, 12], due to the cookies-based authentication of contemporary web browsers.

3.3 Password Managers

An alternative approach for managing authentication using software is Password Managers (PMs) [4, 11, 23]. Using this mechanism, when a user is to register with a service, the PM will store the password for the user, and when they are to log into this service at a later time, the password will be automatically entered by the

PM. Although many interesting PMs have been proposed in the past [4, 11, 23], there are studies [6, 24] questioning the provided security of some of them.

Password Managers need to store all the user passwords, in a safe place, and usually encrypted with a master password. They require the user to only remember one password. Typical places for password storage are local files inside the users' devices, third-party cloud services, or cloud services operated by the password management software company themselves.

Unfortunately, PMs have two basic limitations. First, in case passwords are stored at the user's device, then they have to be synced across multiple devices. Second, passwords stored in third-party databases can be badly maintained, and as a consequence, data breaches may happen, and passwords may be leaked [13]. Although *∅pass* can be viewed as a PM, given that the system manages all passwords, contrary to other managers, it never stores or reuses any password. Therefore, the above reported problems associated with PMs do not apply. In addition to that, *∅pass* does not require the user to remember a single master password.

3.4 One-time passwords

One-time passwords (OTPs) are passwords that are only effective for a fixed period of time, and become invalid after their first usage. The advantage of OTPs is that passwords are invulnerable against spyware (such as key loggers), and replay attacks. In addition, if a single account password is leaked, by using this mechanism, the rest of the accounts will remain safe.

However, there are still some disadvantages to this method. One of them is that the user needs to either maintain a large amount of private keys (for Time-based One-Time Passwords (TOTP)), or, in the case of SMS, provide the website with their phone number, have it available at all times, and manually complete the process for all required logins. In the case of SMS, the website must also incur the cost of messaging the user, which is fixed per login attempt, can be considerable, and may force the website to require SMS authentication less often for it to be sustainable. Costs from the popular SMS service Twilio range from \$0.01 to \$0.10 [2], which can be considered steep to pay for each authentication. Finally, implementing OTP requires changes in the server code, while in the case of *∅pass* this thing is not currently required.

∅pass takes an idea from OTPs that increases security, and that is that passwords should only be used once, and then discarded. With this in mind, every time *∅pass* logs a user in, the password that was set to the database is random, and different from the one used in the previous login.

4 ARCHITECTURE

Based on the four different concepts we discussed in Section 3, we now present, in high-level, the design of our approach. Specifically, *∅pass* is a novel authentication system, which is based on password reminders, and with the help of an e-mail provider that delivers the one-time passwords to the user every time, it authenticate them to a web application or service.

In Figure 1, we illustrate the work-flow of an authentication session when *∅pass* is in place. Consider Alice wanting to log into a web service. *∅pass* triggers a password reminder procedure. Once

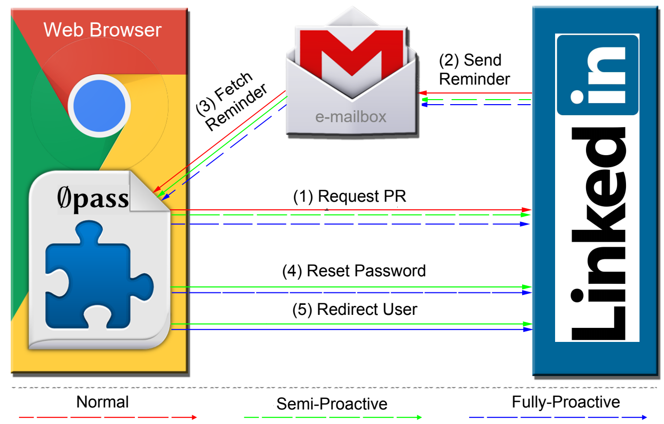

Figure 1: The basic architecture steps of the three modes of *∅pass*. The dashed lines indicate the already completed steps, before the user requests to access the service.

the request has been sent, the service sends an e-mail to Alice's mailbox[1]. *∅pass* polls inbox, checking for an incoming message related to this process. As soon as the e-mail arrives, *∅pass* resets Alice's password with a new random password, using the link included in the e-mail, and after creating the necessary cookies in the browser, it redirects Alice to the homepage of the service.

Once *∅pass* submits a password to a service, then it is immediately discarded from the device's memory. Therefore, all passwords are stored *nowhere* else than in the web service's database, cryptographically hashed. Thus, it can not be leaked, phished, or recycled accidentally by the user across multiple services. Whenever a user needs to re-authenticate with a service, *∅pass* does not provide the existing password, since no one knows it, apart from the web service itself. Instead, *∅pass* initiates a new PR process for recovering the artificially lost password.

Recall that the authentication to a web service is an operation taking place not very frequently; instead users tend to authenticate once, and maintain long sessions, which are handled by cookies, or other browser storage mechanisms [3, 12]. As a consequence, *∅pass* do not need to perform the same procedure every time a user logs in a web service. *∅pass* will be invoked again only when the cookie session expires and user needs to re-authenticate with the service. This phenomenon avoids, in our case, the e-mail provider from learning about *what* the user visits, and *when*, contrary to the SSO alternative. *∅pass* is able to orchestrate several password reminder mechanisms, of different web services, and be able to receive and recognize their e-mails, completing the reset process when required.

4.1 Proactive modes of operation

A careful reader, at this point, may argue that the time a system needs to log its users into their desired web services is crucial for its adoption. Essentially, *∅pass* needs the user to wait for the PR e-mail

[1]Of course, some users may not feel comfortable giving such access to their personal email account. For this reason, *∅pass* suggests the users, instead of their personal email account, to use a secondary disposable email account dedicated for registration purposes only.

delivery before logging them in. In order to reduce this latency, our approach provides two extra modes of operation, namely *semi-proactive* and *fully-proactive*, as depicted in Figure 1.

When the user is to authenticate using the *semi-proactive* mode, the necessary e-mail has already been sent, so ∅*pass* resets the password using this e-mail. When they are to authenticate using the *fully-proactive* mode, the whole procedure has been completed, so ∅*pass* just redirects the user into the service. More specifically, in *semi-proactive* mode, ∅*pass* requests a PR for each service the user is not already logged in, as soon as they starts their browser. Therefore, when the user is to log into one of these services, the prototype will fetch the already sent PR e-mail to reset the password, and complete the authentication. On the other hand, using *fully-proactive* mode, ∅*pass* requests a PR, resets the password, and completes the authentication procedure for each of the services the user is not logged in, when they starts their browser. When the user is to log into a service, they will be just be redirected into the service since the authentication will already be completed.

5 IMPLEMENTATION

In order to explore the feasibility and effectiveness of our approach, we implement a prototype of ∅*pass* as a browser extension. As an example of web service that the user desires to authenticate with, we use LinkedIn. Our approach leverages password reminders and therefore it utilizes the password reset functionality of web services [2]. As a consequence, ∅*pass* needs access to the corresponding registration email of the user, which in our test scenario of LinkedIn this is a Gmail address.

It is important to note at this point, that in ∅*pass* the remote web service (e.g., LinkedIn) acts as a *non-collaborating* service and thus no modifications are needed in the server side. Users have just to install the extension in their browser. They are not required to provide the extension with *any* credentials but only to give read-access permission to the dedicated email account of theirs.

The procedure of our prototype is the following: Whenever a web service is visited, the extension checks if the user is already logged in by verifying the existence and expiration date of the, related with the web service, session cookies. In case of a non-logged in user, ∅*pass* triggers a Password Reminder (PR) procedure to reset the password by email.

In order to have read-access to the user's email inbox and retrieve the corresponding PR email, ∅*pass* uses Atom [1]. Atom is an aggregator for several sources (e.g., RSS, blogs) including mail inboxes. ∅*pass* utilizes Atom's Gmail Inbox Feed (GIF) to retrieve the user's inbox as an XML document. This XML document provides metadata (not full content), including the user's e-mail address, number of unread mails, the timestamp for each email, their title and sender, a summary of content (containing just a small part of the e-mail body), a URL where the ID of the e-mail can be found.

First, ∅*pass* parses the XML output and obtains the user's very own e-mail address needed for the PR. Web services like LinkedIn during Password Reminder provide the user with a form, where the user has to fill in their e-mail address. The extension automates this functionality by directly fill and submit through AJAX-performed `HTTP Requests` the email address form. Next, the extension polls the Atom feed, until the incoming PR e-mail is received. Whenever a new email is received ∅*pass* analyzes the XML provided metadata to identify from the sender and title if the email is the PR email that it waits for. If so, ∅*pass* uses the e-mail ID to fetch the entire body of the e-mail from the email provider. We assume at this point that the user has an active session with his email account so usually no login password for Gmail would be required that could cause additional delays to the authentication operation.

In or our testing case, LinkedIn in its PR e-mails, sends to the users a link to a website to complete the reset procedure by filling up a form with their new password. Since ∅*pass* knows this website's URL, it automatically submits the new password by using a fresh randomly generated token. Then the extension redirects the user to the LinkedIn homepage, and the corresponding cookie is automatically created. After that, the user is successfully authenticated and they can continue browsing the web service.

We implement our prototype in Javascript as an extension for the popular web browser of Google Chrome [3]. In order for ∅*pass* to perform all necessary HTTP requests, it uses the popular AJAX *XMLHttpRequest* Web API [26].

6 EVALUATION

In this section, we evaluate the performance of each internal step of ∅*pass* and the performance gains of its different modes of operation. Then, we compare this performance against the current state-of-the-art authentication mechanism. Finally, we conduct a security evaluation of our approach.

6.1 Performance evaluation

∅*pass* breakdown: ∅*pass*'s operation can be broken down in 5 core steps which include the following:

(1) Initialize: the first step, where ∅*pass* checks if the user is already logged-in by searching the for the associated session cookies in the user's browser.
(2) Request PR: if the user is not logged-in, ∅*pass* requests from the service to reset the password by issuing a Password Reminder (PR) email.
(3) Fetch PR: as a next step, ∅*pass* (by linking with the user's dedicated email client) polls periodically for the appropriate PR email.
(4) Complete Process: once the PR e-mail arrives, ∅*pass* completes the procedure by creating a long random token, which is used as the new password, and submits the final form to reset the user's existing one.
(5) Redirect: Finally, the ∅*pass* redirects the user to the service's webpage, and they can continue browsing as a fully authenticated user.

We run our approach 50 times throughout 1 day and in Figure 2 we plot the average execution time per step. The overall time our approach needs to authenticate a user ranges from 6.2 to 11.7 seconds (8.3 seconds on average).

[2]Some web services impose restrictions on the frequency one can reset a password within a day. In ∅*pass* we use this functionality only upon cookie expiration fully complying with such application limits.

[3]Of course, our extension can be easily ported to other browsers as well.

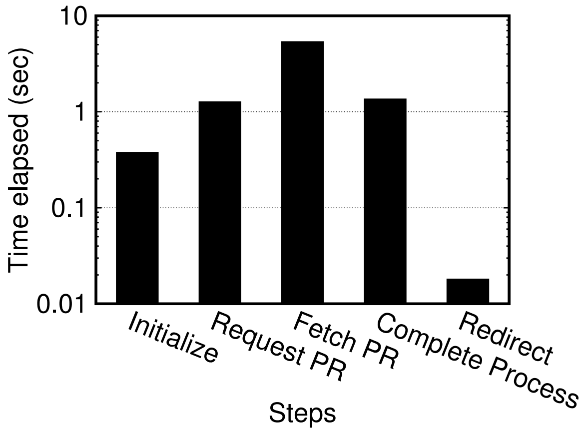

Figure 2: Average execution time for each step of ∅*pass*. As expected, the "Fetch PR" step which includes the PR e-mail transmission takes around 5.3 seconds on average, imposing the higher latency to the overall system's performance than the rest of the steps.

This significant deviation is caused mainly due to the variation of the PR e-mail transmission time of Fetch PR step. It is apparent, hence, that this particular step is the more costly, responsible of the 64% of the overall execution time of ∅*pass*. It's important to note, at this point, that the PR e-mail transmission time highly depends on the users' network speed and location, as well as the remote server's current load.

Furthermore, as expected, we see in Table1 the steps that require communication with the service and HTTP request transmission contributing more to the overall latency (Request PR: 15%, Complete Process: 16%) than the rest local processing steps (Initialize: 4.5%, Redirect: 0.2%).

Proactivity: The overall latency of ∅*pass* (i.e., 8.3 seconds on average) sounds impractical for a user to log into a service. From our experiments, we see that Fetch PR step is able to skyrocket the overall latency overhead of ∅*pass*. To mitigate this issue, as we discussed in section 4.1, we introduce in ∅*pass* two modes of operation. These more proactive modes are able to eliminate the idle times of the system. To achieve that, we pro-actively perform the most time-consuming functionalities before the user requests access to a service. More specifically:

(1) Semi-Proactive: Using Semi-Proactive mode of ∅*pass*, the PR email gets requested and fetched before the moment the user wants to log into a service. This means that ∅*pass* upon login request is ready to move directly to Complete Process step, reset the password, and redirect the user into the service. As we can see in Figure 2, the average time of these steps is only 2.3 seconds.
(2) Fully-proactive: Using the Fully-proactive mode, the whole authentication procedure of ∅*pass* has already been completed asynchronously, as soon as the extension detects the web service's expired session cookies. As a consequence, the moment the user attempts to log into the web service, ∅*pass* simply finalize the password reset procedure redirecting the user to the service. The average time for the entire process in this scenario is 0.015 to 0.020 seconds, significantly improving the overall user experience making the ∅*pass* operation fully transparent.

Step	Portion
Initialize	4.5%
Request PR	15%
Fetch PR	64%
Complete Process	16%
Redirect	0.2%
Overall ∅pass *execution time*	8.3 sec

Table 1: Contribution to the overall execution time of ∅*pass* for the different internal steps.

∅pass Vs. SSO - performance comparison: SSO is one of the state-of-the-art authentication mechanisms to date. In Figure 3, we compare the average execution time for the three modes of ∅*pass*: (i) plain, (ii) semi-proactive, (iii) fully-proactive, and Facebook SSO, namely Facebook Connect, that a user can use to authenticate themselves with LinkedIn. Using Facebook Connect, the time needed for authentication ranges from 0.9 to 3.5 seconds, with an average of 2.3 seconds. This is lower than what the normal mode of ∅*pass* requires and directly comparable with semi-proactive mode. Yet, it is interesting to see that fully-proactive mode takes only 0.018 seconds on average, meaning that it needs only 1% of Facebook SSO's overall time. It is also worth noting that contrary to Facebook's SSO, ∅*pass* is applied to unaware services without requiring *any* modifications on the server side.

6.2 Security Evaluation

Initially, it is important to denote the attacks that are beyond the scope of ∅*pass*. Specifically, in case of attackers able to take control of the user's device, ∅*pass* cannot provide any defense. Of course, an attacker with such capabilities can retrieve the user's passwords in several ways (keylogger, browser session hijacking etc.).

As discussed in Section 4, ∅*pass* collaborates with an e-mail provider (Gmail in our prototype), in order to receive the necessary password reminders. Similar to the above adversarial model, ∅*pass* is unable to defend the user, in case of a very capable attacker able to compromise the entire operation of the e-mail provider (i.e., Google in our case).

On the other hand, ∅*pass* raises the bar against attackers that can leak the provider's password database, phish [7], or real-time phish the credentials of a Gmail account [19]. The user has a pretty strong (computationally hard to crack) e-mail password, and the user protects their password with 2FA. Although, these techniques (strong password, 2FA) are available today, they are not always adopted by users [21], since usability is heavily reduced. With ∅*pass*, we maintain usability, since we need to apply all these only on a single password. In parallel, assuming that 2FA is implemented correctly, for instance by sending code (in the form of a clickable URL) and not just an access token [9], ∅*pass* can protect users even when a real-time phishing attack is in place.

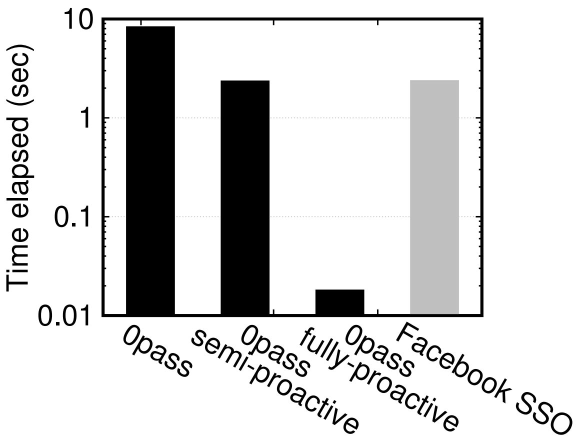

Figure 3: Average execution time of the three different modes of ∅*pass*, compared to Facebook SSO. As we see, although Facebook SSO takes quite less than plain ∅*pass* to authenticate the user, fully-proactive mode of ∅*pass* significantly outperforms Facebook SSO requiring only 1% of its execution time!

Finally, ∅*pass* does not generate authorization tokens as traditional SSOs do. ∅*pass* allows a third party (i.e., the e-mail provider) to *only vet* for the creation of a new user session, exactly as it happens now when the user has lost their password. Unlike SSOs, the e-mail provider does not generate *any* access token that can grant access. Only the user's web browser has the credentials (i.e., a session cookie) to authenticate with the target web site. Notice, that authorization tokens stored in third parties can be leaked exactly as it happens with passwords.

7 CONCLUSION

In this paper, we designed, implemented, and evaluated ∅*pass*: a system that combines ideas from password reminders, single sign-on services, password managers, and one-time passwords, and provides a novel mechanism for user authentication based on server-generated one-time passwords though the well known Password Reminder process. In ∅*pass* no passwords are stored in the user side and the user does not need to memorize anything. As a consequence, passwords cannot be leaked or stolen, and the user receives better protection in phising attacks. We implemented ∅*pass* as a Chrome browser extension, and evaluated it with LinkedIn. Our approach, when compared with Facebook Connect, has a negligible overhead if ∅*pass* operates in a proactive mode.

Acknowledgments

The research leading to these results has received funding from the General Secretariat for Research and Technology (GSRT) of Greece, the Hellenic Foundation for Research and Innovation (HFRI) and European Union's Marie Sklodowska-Curie grant agreement No 690972 (project PROTASIS). The paper reflects only the authors' view and the Agency and the Commission are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Gmail inbox feed. https://mail.google.com/mail/feed/atom.

[2] Twilio pricing page. https://www.twilio.com/sms/pricing/us, 2018.

[3] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009.

[4] H. Bojinov, E. Bursztein, D. Boneh, and X. Boyen. Kamouflage: Loss-resistant password management. In *Proceedings of the 15th European Symposium On Research In Computer Security*, September 2010.

[5] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 553–567, Washington, DC, USA, 2012. IEEE Computer Society.

[6] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *15th USENIX Security Symposium*, USENIX Security, 2006.

[7] S. D'Alfonso. Phishing attacks collect 70 percent of credentials within the first hour. https://securityintelligence.com/phishing-attacks-collect-70-percent-of-credentials-within-the-first-hour/, 2017.

[8] R. Dhamija, J. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, SIGCHI, 2006.

[9] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan. The password reset mitm attack. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 251–267, May 2017.

[10] Google Developers. Google Accounts Authentication and Authorization. https://developers.google.com/accounts/docs/GettingStarted, 2018.

[11] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web*, WWW, 2005.

[12] J. Huang and R. W. White. Parallel browsing behavior on the web. In *Proceedings of the 21st ACM conference on Hypertext and Hypermedia*, 2010.

[13] G. Kontaxis, E. Athanasopoulos, G. Portokalidis, and A. D. Keromytis. Sauth: Protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 187–198, New York, NY, USA, 2013. ACM.

[14] M. Miculan and C. Urban. Formal analysis of facebook connect single sign-on authentication protocol. In *Proceedings of the 37th International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2011.

[15] C. Mills. Mozilla persona project. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona, 2017.

[16] P. Moore. Does two factor authentication actually weaken security? 2015.

[17] D. Morin. Announcing facebook connect. https://developers.facebook.com/blog/post/2008/05/09/announcing-facebook-connect/.

[18] D. Moth. Uk shoppers abandoned over Âč1bn of online transactions in 2011. https://econsultancy.com/blog/9434-uk-shoppers-abandoned-over-1bn-of-online-transactions-in-2011, 2012.

[19] D. Olenick. Massive google docs phishing attack targeted credentials, permissions. https://www.scmagazine.com/massive-google-docs-phishing-attack-targeted-credentials-permissions/article/654938/, 2017.

[20] Openwave Mobility. https://owmobility.com/press-releases/research-shows-wasting-16-billion-hours-year-hunting-passwords/, 2017.

[21] T. Petsas, G. Tsirantonakis, E. Athanasopoulos, and S. Ioannidis. Two-factor authentication: is the world ready?: quantifying 2fa adoption. In *Proceedings of the eighth european workshop on system security*, page 4. ACM, 2015.

[22] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the ACM workshop on Digital Identity Management*, 2006.

[23] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th USENIX Security Symposium*, USENIX Security, 2005.

[24] D. Silver, S. Jana, D. Boneh, E. Chen, and C. Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, San Diego, CA, Aug. 2014. USENIX Association.

[25] S.-T. Sun, Y. Boshmaf, K. Hawkey, and K. Beznosov. A billion keys, but few locks: the crisis of web single sign-on. In *Proceedings of the New Security Paradigms Workshop*. ACM, 2010.

[26] W3Schools. Ajax - the xmlhttprequest object. https://www.w3schools.com/js/js_ajax_http.asp, 2018.

[27] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.