All Your Browser-saved Passwords Could Belong to Us: a Security Analysis and a Cloud-based New Design

Rui Zhao
University of Colorado Colorado Springs
Department of Computer Science
Colorado Springs, CO 80918, USA
rzhao@uccs.edu

Chuan Yue
University of Colorado Colorado Springs
Department of Computer Science
Colorado Springs, CO 80918, USA
cyue@uccs.edu

ABSTRACT

Web users are confronted with the daunting challenges of creating, remembering, and using more and more strong passwords than ever before in order to protect their valuable assets on different websites. Password manager is one of the most popular approaches designed to address these challenges by saving users' passwords and later automatically filling the login forms on behalf of users. Fortunately, all the five most popular Web browsers have provided password managers as a useful built-in feature. Unfortunately, the designs of all those Browser-based Password Managers (BPMs) have severe security vulnerabilities. In this paper, we uncover the vulnerabilities of existing BPMs and analyze how they can be exploited by attackers to crack users' saved passwords. Moreover, we propose a novel Cloud-based Storage-Free BPM (CSF-BPM) design to achieve a high level of security with the desired confidentiality, integrity, and availability properties. We have implemented a CSF-BPM system into Firefox and evaluated its correctness and performance. We believe CSF-BPM is a rational design that can also be integrated into other popular Web browsers.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection.*; H.4.3 [**Information Systems**]: Communications Applications—*Information browsers*; K.4.4 [**Computing Milieux**]: Electronic Commerce—*Security*.

Keywords

Web browser, password manager, security, phishing, cloud

1. INTRODUCTION

Text-based passwords still occupy the dominant position in online user authentication [7, 16, 17]. They protect online accounts with valuable assets, and thus have been continuously targeted by various cracking and harvesting attacks. Password security heavily depends on creating strong passwords and protecting them from being stolen. However, researchers have demonstrated that strong passwords that are sufficiently long, random, and hard to crack by attackers are often difficult to remember by users [5, 12, 22, 24, 36]. Meanwhile, no matter how strong they are, online passwords are also vulnerable to harvesting attacks such as phishing [19, 27, 39, 40]. These hard problems have been further aggravated by the fact that Web users have more online accounts than before and are forced to create and remember more and more usernames and passwords probably using insecure practices such as sharing passwords across websites [13, 30].

Password manager, particularly Browser-based Password Manager (BPM) is one of the most popular approaches that can potentially best address the online user authentication and password management problems. Browser integration enables BPMs to easily save users' login information including usernames and passwords into a database, and later automatically fill the login forms on behalf of users. Therefore, users do not need to remember a large number of strong passwords; meanwhile, BPMs will only fill the passwords on the login forms of the corresponding websites and thus can potentially protect against phishing attacks.

Fortunately, all the five most popular browsers Internet Explorer, Firefox, Google Chrome, Safari, and Opera have provided password managers as a useful built-in feature. Unfortunately, the designs of all those BPMs have severe security vulnerabilities. In essence, our key observation is that the encrypted passwords stored by those BPMs are very weakly protected – they can be trivially decrypted by attackers for logging into victims' accounts on the corresponding websites. We have developed tools to demonstrate that once stolen, the encrypted website login information saved by the five browsers (without using a master password in Firefox and Opera) can all be easily decrypted by attackers. When a master password is used in Firefox or Opera, even though decrypting a user's login information becomes harder, brute force attacks and phishing attacks against the master password are still quite possible.

In this paper, we uncover the vulnerabilities of existing BPMs and analyze how they can be exploited by attackers to crack users' saved passwords. Moreover, we propose a novel Cloud-based Storage-Free BPM (CSF-BPM) design to achieve a high level of security with the desired confidentiality, integrity, and availability properties. CSF-BPM is cloud-based storage-free in the sense that the protected data will be completely stored in the cloud – nothing needs to be stored on a user's computer. We want to move the storage

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

into the cloud for two main reasons. One is that in the long run trustworthy storage services in the cloud [8, 25] can better protect a regular user's data than a local computer. The other is that the stored data can be easily accessible to the user across different OS accounts and different computers.

We have followed standard responsible disclosure practices and reported those vulnerabilities to the respective browser vendors. We have implemented a CSF-BPM system and seamlessly integrated it into Firefox. We have evaluated the correctness and performance of this system. We believe CSF-BPM is a rational design that can also be integrated into other popular browsers to make the online experience of Web users more secure, convenient, and enjoyable.

2. RELATED WORK AND BACKGROUND

In this section, we review the related work and the background information of BPMs in the five most popular browsers.

2.1 Related Work

Researchers pointed out long ago that weak passwords suffer from brute-force and dictionary attacks [12, 24]. However, strong passwords that are sufficiently long, random, and hard to crack by attackers are often difficult to remember by users due to human memory limitations [5, 36]. Recently, Florêncio and Herley performed a large-scale study of Web password habits and demonstrated the severity of the security problems such as sharing passwords across websites and using weak passwords [13]. A large-scale user study recently performed by Komanduri et al. demonstrated that many Web users write down or otherwise store their passwords, especially those higher-entropy passwords [22].

To help users better manage their online accounts and enhance their password security, researchers have proposed many solutions such as password managers [34], single sign-on systems [23, 31], graphical passwords [11, 32], and password hashing systems [15, 28, 38]. As analyzed in Section 1, password managers especially BPMs have the great potential to best address the challenges of using many strong passwords and protecting against phishing attacks. Web Wallet [34] is an anti-phishing solution and is also a password manager that can help users fill login forms using stored information; but as pointed out by the authors, users have a strong tendency to use traditional Web forms for typing sensitive information instead of using a special browser sidebar user interface. The security concerns of single sign-on systems are analyzed in [23], and the business model limitations are analyzed in [31]. Security limitations of graphical passwords are analyzed in [11, 32]. Security and usability limitations of password hashing systems are analyzed in [10, 15]. We do not advocate against any of these other approaches. We simply focus on the BPM security in this paper.

2.2 Password Manager Feature of Browsers

Table 1 lists the basic information on the BPM feature of the recent versions of the five most popular Web browsers. We can see that the BPM feature configuration locations are very different among browsers. Indeed, the feature configuration interfaces shown on those locations are also very different among browsers in terms of the configuration options and functions. The BPM feature is enabled by default in four browsers but not in Safari. Only Firefox and Opera employ a master password mechanism, which is, however, not enabled by default so users may not be aware of its importance. Only Firefox and Google Chrome provide a password synchronization mechanism that can allow users to access the saved passwords across different computers.

Table 1: Basic information of BPMs in five browsers.

Browser	Configuration Location	Enabled by Default	Master Password	Password Sync.
Internet Explorer (9.0)	Internet options → Content → AutoComplete Settings → User names and passwords on forms	Yes	No	No
Firefox (15.0)	Options → Security → Passwords	Yes	Yes	Yes
Google Chrome (21.0)	Settings → Show advanced settings... → Passwords and forms	Yes	No	Yes
Safari (5.1)	Preferences → AutoFill → User names and passwords	No	No	No
Opera (12.02)	Settings → Preferences → Forms → Password Manager	Yes	Yes	No

In terms of the dynamic behavior, the interfaces for triggering the remembering and autofill of passwords are inconsistent among browsers. For example, all the browsers display a dialog box to ask a user whether the entered password for the current website should be saved. Only the dialog boxes displayed by Firefox, Google Chrome, and Opera are associated with the address bar, thus technically hard to be spoofed. For another example, Internet Explorer, Firefox, and Opera require a user action before auto-filling the password value on a website; however, Google Chrome and Safari autofill the username and password values once a user visits a login webpage, providing more opportunities for malicious JavaScript to manipulate the login form and information.

3. VULNERABILITY ANALYSIS

In this section, we define the threat model and assumptions, use an analogy to justify the essential problem of existing BPMs, and provide a detailed vulnerability analysis.

3.1 Threat Model and Assumptions

"Where a threat intersects with a vulnerability, risk is present [3]." For BPMs, the threat sources are attackers who want to steal the stored sensitive login information. Our basic threat model is that attackers can temporarily install malware such as Trojan horses and bots on a user's computer using attacks such as drive-by downloads [26, 33]. The installed malware can then steal the login information stored by BPMs. For example, Stone-Gross et al. inferred that 38% of the credentials stolen by the Torpig bot were obtained from the password managers of browsers, rather than by intercepting actual login sessions [30]. Indeed, if the occurrences of such threats are rare or do not have high impacts, BPMs would not bother to encrypt their stored passwords in the first place. Therefore, we do not intend to further identify threat sources, but focus on investigating the vulnerabilities of BPMs that could be exploited by potential threat sources to easily decrypt the stored passwords.

We assume that the installed malware cannot further compromise the operating system to directly identify cryptographic keys from a computer's memory [14] because this identification often requires elevated privilege and is prone to false positives. We assume that the installed malware can be removed from the system in a timely manner, so that

even though sensitive login information stored by BPMs can be stolen within a short period of time, attackers cannot use tools such as keyloggers to further intercept users' passwords for a long period of time. This assumption is reasonable. One typical example is that anti-malware programs such as Microsoft Forefront Endpoint Protection may detect the infection, report the suspicious file transmission, and finally remove the malware and infected files. Another typical example is that solutions such as the Back to the Future framework [18] can restore the system to a prior good state and preserve system integrity. We also assume (as assumed in all the BPMs) that domain name systems are secure and reliable, and we do not specifically consider pharming attacks.

3.2 The Essential Problem and An Analogy

The essential problem is that the encrypted passwords stored by BPMs of the five most popular browsers are very weakly protected. In our investigation, we found without the protection of a master password mechanism, the encrypted passwords stored by the five BPMs (Table 1) can be trivially decrypted by attackers for logging into victims' accounts on the corresponding websites. We have developed tools and verified this severe vulnerability of the latest versions of the five BPMs on Windows 7. This vulnerability is common to all these browsers because the keys used by these browsers for encrypting/decrypting a user's login information can be easily extracted or generated by attackers. The decrypted login information can be easily sent out to attackers and the entire attack could be finished in one second. In the cases when a master password is used by a user in Firefox or Opera (Table 1), even though decrypting a user's login information becomes harder, brute force attacks and phishing attacks against the master password are still quite possible.

Before taking readers into the details of the vulnerabilities in the next two subsections, we want to use an analogy to justify the problem. Suppose we use a safe to store valuables in a home. There is no guarantee that burglars cannot enter the home, but should we allow burglars to easily figure out the combination to open the safe? If the answer is "Yes", why should we bother to use a safe in the first place? A computer (or its OS) is similar to a home, a BPM is similar to a safe because it stores valuable online account information, and a master password is similar to the combination to the safe. Similarly, there is no guarantee that attackers cannot remotely break into the computer [26, 30, 33], but should we allow attackers to easily decrypt the passwords saved by a BPM? If the answer is "Yes", why should a BPM bother to encrypt the saved passwords in the first place?

The current reality is that the "safe" of Google Chrome, Internet Explorer, and Safari does not allow a user to set a "combination" at all. Our decryption tools can easily and accurately open the "safe". Firefox and Opera allow a user to set a "combination", but they do not make it mandatory. Our decryption tools can also easily and accurately open the "safe" of Firefox and Opera if a "combination" was not set. For example, using drive-by downloads, an attacker can deliver our decryption tools to a user's computer and trigger their execution. In one second, all the passwords and usernames saved by BPMs can be completely decrypted and sent back to the attacker's website or email account. The malware detector installed on the user's computer may report suspicious activities, and the user may immediately take actions to disable the Internet connection. But it could be too late! With a successful drive-by download, attackers can perform many types of malicious activities. However, similar to burglars, if attackers know they can easily open the "safe", they would like to first steal the most valuable items from the "safe" within a short period of time.

3.3 Without a Master Password Mechanism

Through source code analysis, binary file analysis, and experiments, we found both Firefox and Opera use the three-key Triple-DES algorithm to encrypt a user's passwords for different websites. ***Firefox*** saves each encrypted username, encrypted password, and plaintext login webpage URL address into the *login* table of an SQLite database file named *signons.sqlite*. The Triple-DES keys are generated once by Firefox and then saved into a binary file named *key3.db* starting from the byte offset location 0x2F90. Although the keys generated on different computers are different, they are not bound to a particular computer or protected by other mechanisms. Therefore, as verified by our tools, an attacker can simply steal both the *signons.sqlite* file and the *key3.db* file and then accurately decrypt every username and password pair on any computer.

Opera encrypts each username and password pair together with the login webpage URL address, and then saves both the ciphertext and Triple-DES keys into a binary file named *wand.dat*. Therefore, this file is self-contained; an attacker can simply steal it and then decrypt every username and password pair on any computer. We have reverse engineered *wand.dat* and developed a tool that can accurately perform such attacks. In more details, this tool identifies special block structures starting from the byte offset location 0x24 in the file, constructs the triple-DES keys saved in each block structure, and finally decrypts the ciphertext saved in each block structure.

In their latest Window 7 versions, all the other three browsers Internet Explorer, Google Chrome, and Safari use the Windows API functions *CryptProtectData* and *CryptUnprotectData* to perform encryption and decryption, respectively. The key benefit of using these two functions is that typically, the encrypted data can only be decrypted under the same Windows user account. To use these two API functions, an application (e.g., a browser) does not generate or provide encryption/decryption keys because the symmetric keys will be deterministically generated in these two functions based (by default) on the profile of the current Windows user. An application can use the *dwFlags* input parameter to specify that the keys should be simply associated with the current computer; it can also use the *pOptionalEntropy* input parameter to provide additional entropy to the two functions.

We found ***Google Chrome*** saves each plaintext username, encrypted password, and plaintext login webpage URL address into the *logins* table of an SQLite database file named *Login Data*. Google Chrome does not provide additional entropy to the two API functions. ***Safari*** saves each plaintext username, encrypted password, and plaintext login webpage URL address into a special property list file named *keychain.plist*. Safari provides a static 144-byte salt as the additional entropy to the two API functions. ***Internet Explorer*** encrypts each username and password pair and saves the ciphertext as a *value data* under the Windows registry entry: "HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\IntelliForms\Storage2\". Each saved *value*

data can be indexed by a *value name*, which is calculated by hashing the login webpage URL address. Internet Explorer also provides the login webpage URL address as the additional entropy to the two API functions.

We found all these three browsers set the *dwFlags* input parameter value as the default value zero, which means that the symmetric keys are associated with each individual Windows 7 user. Therefore, it is not very easy for attackers to decrypt the stolen ciphertexts on another computer or using another Windows account. However, attackers can simply decrypt the stolen ciphertexts on the victim's machine when the victim is logged into the Windows; then, the decrypted login information can be directly sent back to attackers. We have developed tools that can decrypt the ciphertexts stored by all these three browsers. In more details, for Google Chrome, our tool selects each record from the *logins* table of the *Login Data* SQLite database, converts the encrypted password from the SQLite BLOB type to a string type, and supplies the encrypted password to the *CryptUnprotectData* function. For Safari, our tool converts the *keychain.plist* property list file to an XML document, parses the XML document to obtain each encrypted password, and supplies the encrypted password and that static 144-byte salt to the *CryptUnprotectData* function. For Internet Explorer, our tool hashes the popular login webpage URL addresses contained in a dictionary, queries the Windows registry using each hashed URL address to identify a matched *value name*, and supplies the associated *value data* and the corresponding login webpage URL address (as the additional entropy) to the *CryptUnprotectData* function.

3.4 With a Master Password Mechanism

The BPMs of Firefox and Opera allow a user to set a master password (Table 1) to further protect the encryption keys or encrypted passwords. In ***Firefox***, the master password and a global 160-bit salt will be hashed using a SHA-1 algorithm to generate a master key. This master key is used to encrypt those three Triple-DES keys before saving them to the *key3.db* file. Firefox also uses this master key to encrypt a hard-coded string "password-check" and saves the ciphertext to the *key3.db* file; later, Firefox will decrypt this ciphertext to authenticate a user before further decrypting the three Triple-DES keys. In ***Opera***, a master key is generated based on the master password and a 128-bit salt. The salt itself is encrypted using the master password and saved to a specific file named *Opcert6.data*. The master key and Triple-DES keys will be used to double-encrypt each website password. Finally, the ciphertext including the encrypted login webpage URL address, encrypted username, and double-encrypted password will be saved to the *wand.dat* file.

Using a master password can better protect the stored passwords in both browsers. However, the master password mechanism should be carefully designed to maximize security. One main security concern is the brute force attacks against the master password. For one example, if the computation time for verifying a master password is very small as in Firefox and Opera (both of them reject an invalid master password in milliseconds), it is still possible to effectively perform brute force attacks against a user's master password. For another example, encrypting the hard-coded "password-check" string in Firefox for user authentication does not increase security and may actually decrease security in the case when both the *signons.sqlite* file and the *key3.db* file are stolen. Although decrypting the Triple-DES keys is still very difficult if the master password is unknown, an attacker can simply bypass this user authentication step using an instrumented Firefox. Moreover, this hard-coded plaintext and its ciphertext encrypted by the master key can also be used by an attacker to verify the correctness of dictionary or brute-force attacks against the master password.

Another main security concern is the phishing attacks against the master password. For example, attackers can use the JavaScript *prompt()* function to create fake master password entry dialog boxes, which can be displayed by any webpage on all the five browsers without being blocked by browsers' "block pop-up windows" options because they are not in a separate HTML document. We speculate that even such a simple spoofing technique can effectively obtain master passwords from vulnerable users. Indeed, a malicious webpage can also use JavaScript and CSS (Cascading Style Sheets) to create sophisticated dialog boxes that are more similar to genuine master password entry dialog boxes.

4. CSF-BPM DESIGN

We now present the design of CSF-BPM. It is cloud-based storage-free in the sense that the protected data will be completely stored in the cloud – nothing needs to be stored on a user's computer. We want to move the storage into the cloud for two key reasons. One is that in the long run trustworthy storage services in the cloud [8, 25] can better protect a regular user's data than a local computer. The other is that the stored data can be easily accessible to the user across different OS accounts on the same computer and across computers at different locations. This design differs from the BPM designs of all the five most popular browsers. Based on the threat model and assumptions defined in the last section, we design CSF-BPM to synthesize the desired security properties such as confidentiality, integrity, and availability.

4.1 High-level Architecture

Figure 1 illustrates the high-level architecture of CSF-BPM. The BPM of the browser simply consists of a User Interface (UI) component, a Record Management (RM) component, a Record Generation (RG) component, a Record Decryption (RD) component, and a record synchronization (Sync) component. The UI component will provide configuration and management interfaces accessible at a single location. The BPM itself does not include any persistent storage component such as a file or database; instead, it will generate Encrypted Login Information Records (ELIRs), save *protected ELIRs* to a Secure and Reliable Storage (SRS) service in the cloud, and retrieve protected ELIRs in real-time whenever needed. Such a generic BPM design can be seamlessly integrated into different browsers.

An SRS service simply needs to support user authentication (e.g., over HTTPS) and per-user storage so that its deployment in the cloud can be easily achieved. For example, the synchronization service associated with Firefox or Google Chrome (Table 1) could be directly used as an SRS service without making any modification. The SRS service will store a Per-User Protected ELIRs (PUPE) data object (to be illustrated in Figure 2) for each SRS user. The communication protocol between the BPM and SRS is also very simple: after a user is authenticated to SRS, the Sync component of BPM will transparently send HTTPS requests to SRS to *retrieve* or *save* the protected ELIRs of the user. An

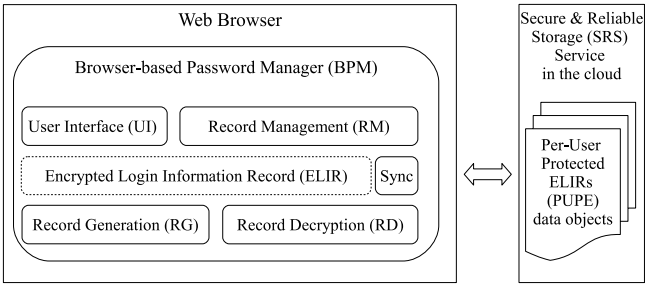

Figure 1: High-level architecture of the Cloud-based Storage-Free BPM (CSF-BPM).

SRS service should be highly reliable and available. However, to further increase reliability and availability, the BPM can store protected ELIRs to multiple independent SRS services. One of them is used as the primary SRS service; others will be used as secondary SRS services. The Sync component of BPM will transparently synchronize protected ELIRs from the primary SRS service to secondary SRS services.

4.2 Design Details

To use CSF-BPM, a user needs to remember a Single Strong Master Password (SSMP) with the strength [4] assured by a proactive password checker and certain length requirement [6, 21, 37]. The user also needs to set up an account (srsUsername, srsPassword) on an SRS service and configure this service once through the UI component of BPM. At the beginning of each browsing session, the user needs to authenticate to the SRS service and provide the SSMP to BPM. After that, BPM will take care of everything else such as triggering the remembering of website passwords, encrypting and decrypting ELIRs, and triggering the autofill of passwords. Both the srsUsername and srsPassword pair and SSMP need be provided only once in a session through the UI component of BPM, and users will be explicitly informed of such a usage requirement. These design choices could be helpful in protecting SSMP against phishing attacks.

The basic format of an ELIR record is shown in Table 2. Here, recordSalt is a large and sufficiently random per-record salt generated by BPM. It is used to calculate the symmetric record key (denoted recordKey) for encrypting a user's plaintext password (denoted sitePassword) for an account (denoted siteUsername) on a website (with siteURL as the login webpage URL address). The recordKey can be deterministically generated by using a password-based key derivation function such as PBKDF2 specified in the PKCS5 specification version 2.0 [20]. The basic format of an ELIR record can also include the IDs (or names) of the username and password fields in the login webpage, and it can be further extended if necessary.

Table 2: The basic format of an ELIR record.

siteURL	siteUsername	encryptedSitePassword	recordSalt	...

Using PBKDF2 [20], our SSMP-based key derivation and password encryption process consists of five steps illustrated in Formulas 1, 2, 3, 4, and 5. The input parameters mainSalt and aeSalt in Formulas 1 and 2 are large and sufficiently random per-user salts generated by BPM at the first time when a user authenticates to the SRS service through the UI component of BPM. In Formulas 1, 2, and 3, the input parameters c1, c2, and c3 represent iteration counts for key stretching; the input parameters dkLen1, dkLen2, and dkLen3 represent lengths of the derived keys, and they are related to the underlying pseudorandom function used in the PBKDF2 implementation.

$$mainKey = PBKDF2(SSMP, mainSalt, c1, dkLen1) \quad (1)$$

$$aeKey = PBKDF2(mainKey, aeSalt, c2, dkLen2) \quad (2)$$

$$recordKey = PBKDF2(mainKey, recordSalt, c3, dkLen3) \quad (3)$$

$$encryptedSitePassword = E(recordKey, sitePassword) \quad (4)$$

$$protectedELIRs = AE(aeKey, concatenatedELIRs) \quad (5)$$

The salts and iteration counts in PBKDF2 are used to secure against dictionary and brute-force attacks, and they need not be kept secret [20]. The strength of SSMP also helps secure against these two types of attacks. In Formula 1, a mainKey is calculated and will be used in each browsing session. SSMP is typed only once and will be erased from memory after mainKey is calculated. In Formula 3, a unique recordKey is generated (using the per-record recordSalt) for each website account of the user. In Formula 4, a NIST-approved symmetric encryption algorithm E such as AES [1] (together with a block cipher mode of operation if the sitePassword is long) can be used to encrypt the sitePassword. In Formula 5, a NIST-approved Authenticated Encryption block cipher mode AE such as CCM (Counter with CBC-MAC) [2] can be used to simultaneously protect confidentially and authenticity (integrity) of the concatenatedELIRs of an SRS user.

The iteration count c1 used in Formula 1 should be large so that the mainKey calculation will take a few seconds; therefore, brute force attacks against SSMP become computationally infeasible. But c1 should not be too large to make a user wait for a long period of time at the beginning of a session. Iteration counts c2 and c3 should not be too large so that generating aeKey and recordKey would not cause a user to perceive any delay. The mainKey is kept in memory in the whole browsing session, while aeKey and recordKey are scrubbed immediately after use. We assumed in Section 3 that attackers cannot identify cryptographic keys from memory. Countermeasures [14] can be taken to further protect this single mainKey in memory. Although Formula 5 will simultaneously protect confidentially and authenticity (integrity) of the concatenatedELIRs of an SRS user, encrypting each sitePassword in Formula 4 is still important. This is because the concatenatedELIRs is also kept in memory in the whole browsing session. In comparison with the mainKey which is basically a random-looking value, the structure of ELIR records and concatenatedELIRs can be easily identified from memory. Therefore, assuming an attacker cannot identify the mainKey but can identify ELIR records from memory, it is still computationally infeasible for the attacker to crack each individual recordKey and decrypt the corresponding sitePassword.

Overall, all the computations including salt generation, key derivation, encryption, and decryption etc. are performed on BPM. Neither the SSMP nor any derived cryptographic key will be revealed to an SRS service or a third

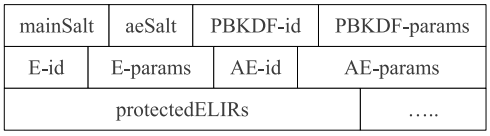

Figure 2: The Per-User Protected ELIRs (PUPE) data object saved for each SRS user.

party. An SRS service does not need to provide any special computational support to BPM; it simply needs to save a PUPE data object for each SRS user.

As illustrated in Figure 2, each PUPE object contains the protectedELIRs (Formula 5) of an SRS user and all the algorithm related information. Here, PBKDF-id specifies the identifier for the PBKDF2 key derivation function [20]; PBKDF-params specify the PBKDF2 parameters such as c1, c2, c3, dkLen1, dkLen2, and dkLen3 used in Formulas 1, 2, and 3. E-id and E-params specify the identifier and parameters, respectively, for the symmetric encryption algorithm (and the mode of operation) used in Formula 4. AE-id and AE-params specify the identifier and parameters, respectively, for the authenticated encryption block cipher mode used in Formula 5. For example, if AE-id specifies the CCM authenticated encryption block cipher mode [2], then AE-params will contain the Nonce and the Associated Data input parameters used by CCM. Each PUPE data object can be simply saved as a binary or encoded string object for an SRS user because its structure does not need to be known or taken care of by any SRS service. Such a PUPE data object design makes the selection of algorithms and the selection of SRS services very flexible.

The iteration counts c1, c2, and c3 can be flexibility adjusted by BPM with or without user intervention to maximize security while minimizing inconvenience to users [9]. In our current design, CSF-BPM adaptively computes the maximum values of iteration counts based on the specified computation times for Formulas 1, 2, and 3, respectively. For example, if a 10-second computation time is specified for deriving the mainKey, CSF-BPM will run Formula 1 for 10 seconds to derive the mainKey and meanwhile finalize the corresponding c1 value. Such a scheme allows CSF-BPM to easily maximize the security strength of key derivation within a specified delay limit on each individual computer.

To decrypt the saved website passwords for autofill, BPM will perform five steps: (1) retrieve the PUPE data object saved for the SRS user; (2) generate the mainKey and aeKey using Formulas 1 and 2; (3) decrypt and verify the protectedELIRs using the reverse process of Formula 5 such as the CCM Decryption-Verification process [2]; (4) obtain the recordSalt of each ELIR and generate the recordKey using Formula 3; (5) finally, decrypt the encryptedSitePassword using the reverse process of Formula 4. Note that at step (3), both the integrity of the protectedELIRs and the authenticity of the BPM user are verified because the success of this step relies on using the correct SSMP. Also at this step, siteURL and siteUsername of all the ELIRs can be obtained by BPM to determine whether this user has previously saved login information for the currently visited website. Normally, the first three steps will be performed once for the entire browsing session, and the last two steps will be performed once for each currently visited website that the user has previously saved login information for it.

Because all the salts are randomly generated by BPM, the protectedELIRs saved to different SRS accounts or different SRS services will be different. BPM can transparently change mainSalt, aeSalt, and every recordSalt whenever necessary. A user also has the flexibility to change SSMP and any sitePassword whenever necessary. In these cases, all what need to be done by BPM is to update the new PUPE data object and ELIRs to each corresponding SRS service account. A user can also flexibly change any srsPassword, which is completely independent of SSMP.

4.3 Design Rationales and Security Analysis

In terms of the confidentiality, first, by having a unique cloud-based storage-free architecture, CSF-BPM can in the long run effectively reduce the opportunities for attackers to steal and further crack regular users' saved passwords. Second, even if attackers (including insiders of an SRS service) can steal the saved data, it is computationally infeasible for attackers to decrypt the stolen data to obtain users' login information for different websites. CSF-BPM provides this security guarantee by mandating a strong SSMP that satisfies certain strength requirements [6, 21, 37], by using the PBKDF2 key derivation function [20] with randomly generated salts and adaptively computed large iteration counts, and by following NIST-approved symmetric encryption [1] and authenticated encryption [2] algorithms. Basically, even if attackers can steal the saved data, they have to guess a user's strong SSMP in a very large space (determined mainly by the length and character set requirements of SSMP) with each try taking seconds of computation time.

In terms of the integrity, the NIST-approved CCM authenticated encryption algorithm [2] enables CSF-BPM to accurately detect both any invalid SSMP try and any modification to a saved PUPE data object. Moreover, this detection is securely performed in the sense that attackers cannot take advantage of it to effectively conduct brute force attacks against the SSMP.

In terms of the availability, an SRS simply needs to be a storage service in the cloud and it does not need to provide any special computational support. Such a design decision makes it very easy to either use an existing storage service in the cloud as an SRS service or deploy a new SRS service by an organization. CSF-BPM supports multiple SRS services and it uses a simple HTTPS-based communication protocol; these design decisions also further enhance the availability.

CSF-BPM offers a better security in comparison with the BPMs of Firefox and Opera that also provide a master password mechanism. Firefox and Opera save the encrypted data locally on a user's computer and they do not use strong key derivation functions (Section 3.4); thus, their confidentiality assurance is weak in consideration of brute force attacks. Both Firefox and Opera can detect an invalid master password try, but the detection mechanism is not secure (Section 3.4). Both browsers do not detect any modification to the saved data; the modified data will still be decrypted into incorrect and often non-displayable values, but no information is provided to a user. Opera does not provide a password synchronization mechanism, while the Sync mechanism of Firefox is tightly bound to Mozilla's own server [41]; thus, the availability of the saved data is not well assured by these two BPMs.

Other cloud-based password system design alternatives also exist, but they often have different design objectives and limitations. For example, Passpet [38] can help a user generate passwords for different websites based on a master

password. Similar to Password Multiplier [15], Passpet is essentially a password generator instead of a password manager because it uses password hashing techniques to deterministically generate website passwords instead of remembering users' original passwords. Requiring users to migrate their original passwords to hashed passwords is a biggest limitation of hashing-based password generation solutions as acknowledged in the Password Multiplier paper [15]. In addition, Passpet imposes very special requirements on its remote storage server: the SRP authentication protocol [35] must be used and some specific commands must be supported. These requirements limit the deployability of Passpet.

5. IMPLEMENTATION AND EVALUATION

CSF-BPM is designed in a generic way so that it is implementable in different Web browsers and meanwhile it can easily use different SRS services. We have implemented a CSF-BPM system using JavaScript and seamlessly integrated it into the Firefox Web browser. This system can directly use the Firefox Sync server operated by Mozilla [41] as an SRS service without making any modification to this server; thus, a free SRS service is directly available to users. Note that in our implementation, the interfaces for triggering the remembering and autofill of passwords in Firefox (Section 2.2) are not changed; only the operations happen behind the scenes are changed. Currently, we used the PBKDF2 [20], CCM [2], and AES [1] implementations provided in the Stanford JavaScript Crypto Library [29].

We made some modifications to the Sync module in Firefox. In the original Sync module, a 26-character Sync key is generated when a user creates a Sync account. This Sync key is not shared with the Firefox Sync server, and is mainly used to protect other cryptographic keys that are stored on the Firefox Sync server for a user. A user must save this Sync key and provide it to Firefox on different computers together with the Sync account username and password whenever the Sync mechanism needs to be used. This requirement limits the usability of the Sync mechanism. The Sync key is not needed at all in the CSF-BPM design. Therefore, in our implementation, one main modification to the Sync module is removing the dependence of using Firefox Sync server on Sync key for the password manager feature. As a result, a user does not need to save and provide the Sync key at all if he or she uses CSF-BPM and uses the Firefox Sync server as the SRS service. The other main modification is that we use a Weave Basic Object (WBO) [41] assigned to the default Mozilla *passwords collection* to store the PUPE object in the Firefox Sync server. Both modifications are specific to using the Firefox Sync server as the SRS service.

We have built the Firefox version CSF-BPM on a Ubuntu Linux system. We have evaluated the correctness of our implementation and its integration with Firefox. We selected 30 popular or our frequently used websites to perform the evaluation. On each website, we (1) opened Firefox and typed an SRS account (i.e., a Firefox Sync account) and SSMP; (2) logged into the website and confirmed to save the website password; (3) logged out the website and logged into it again with the auto-filled password; and (4) finally closed Firefox, re-opened Firefox, typed the SRS account and SSMP, and logged into the website again with the auto-filled password. By performing those steps, we verified that our implementation works precisely as designed; meanwhile, it integrates smoothly with Firefox and does not cause any logic or runtime error.

We have conducted both micro-benchmark experiments and macro-benchmark experiments to evaluate the performance of CSF-BPM. In those experiments, we ran CSF-BPM on a desktop computer with 2.33GHz CPU, 3.2 GB memory, and 100Mbps network card. In our experiments, we evaluated the performance of key derivation, password encryption and decryption, concatenatedELIRs encryption and decryption, and PUPE upload and retrieval. Overall, all those operations were efficiently performed, and we did not observe any noticeable delay.

6. DISCUSSION

We analyzed in Section 4 that CSF-BPM provides a high level of security. We now further discuss the usability and limitations of CSF-BPM. To be fair, we compare the usability of our Firefox version CSF-BPM with that of the combined Firefox master password mechanism and Sync mechanism. As mentioned in Section 5, our implemented CSF-BPM only changes the operations happening behind the scenes; therefore, its overall usability is comparable to that of the combined Firefox master password mechanism and Sync mechanism. Moreover, our implemented CSF-BPM has one obvious usability improvement because it uses the Firefox Sync server as an SRS service but does not require a user to save and provide the 26-character Sync key generated by Firefox. In CSF-BPM, the PUPE object is already protected, thus the Sync key is no longer needed at all.

In terms of limitations, first, our current CSF-BPM is implemented in JavaScript. The security and performance of CSF-BPM can be further improved if those cryptographic algorithms are implemented in C++. Currently, we are working on implementing those cryptographic algorithms into a Firefox XPCOM (Cross Platform Component Object Model) component using C++. Second, if a CSF-BPM user forgets the SSMP, all the passwords saved on SRS services cannot be correctly decrypted. Therefore, remembering the SSMP becomes very important for CSF-BPM users. However, remembering an SSMP should be much easier than remembering many strong passwords for different websites. Third, at the beginning of a Web browsing session, a user must wait for a few seconds so that CSF-BPM can complete the mainKey derivation. However, once the mainKey is derived, password remembering and autofill operations can be smoothly performed by users as usual.

7. CONCLUSION

In this paper, we uncovered the vulnerabilities of existing BPMs and analyzed how they can be exploited by attackers to crack users' saved passwords. Moreover, we proposed a novel Cloud-based Storage-Free BPM (CSF-BPM) design to achieve a high level of security with the desired confidentiality, integrity, and availability properties. We implemented a CSF-BPM system and seamlessly integrated it into the Firefox Web browser. We evaluated the correctness and performance of this system. Our evaluation results demonstrate that CSF-BPM can be efficiently used to manage online passwords. We believe CSF-BPM is a rational design that can also be integrated into other popular Web browsers to make the online experience of Web users more secure, convenient, and enjoyable.

8. REFERENCES

[1] Advanced Encryption Standard (AES). In *NIST FIPS 197*, 2001.

[2] The CCM Mode for Authentication and Confidentiality. In *NIST SP 800-38C*, 2004.

[3] Information Security Handbook: A Guide for Managers. In *NIST SP 800-100*, 2007.

[4] Electronic Authentication Guideline. In *NIST SP 800-63-1*, 2011.

[5] A. Adams and M. A. Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, 1999.

[6] M. Bishop and D. V. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, 1995.

[7] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. of IEEE S&P*, 2012.

[8] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proc. of CCS*, 2009.

[9] X. Boyen. Halting password puzzles: hard-to-break encryption from human-memorable keys. In *Proc. of USENIX Security Symposium*, 2007.

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

[11] D. Davis, F. Monrose, and M. K. Reiter. On user choice in graphical password schemes. In *Proc. of USENIX Security Symposium*, 2004.

[12] D. C. Feldmeier and P. R. Karn. Unix password security – ten years later. In *Proc. of CRYPTO*, 1989.

[13] D. Florêncio and C. Herley. A large-scale study of web password habits. In *Proc. of WWW*, 2007.

[14] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. of USENIX Security Symposium*, 2008.

[15] J. A. Halderman, B. Waters, and E. W. Felten. A convenient method for securely managing passwords. In *Proc. of WWW*, 2005.

[16] C. Herley and P. C. van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36, 2012.

[17] C. Herley, P. C. van Oorschot, and A. S. Patrick. Passwords: If we're so smart, why are we still using them? In *Proc. of FC*, 2009.

[18] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *Proc. of ACSAC*, 2006.

[19] M. Jakobsson and S. Myers. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley-Interscience, ISBN 0-471-78245-9, 2006.

[20] B. Kaliski. RFC 2898, PKCS5: Password-Based Cryptography Specification Version 2.0, 1999.

[21] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. of IEEE S&P*, 2012.

[22] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *Proc. of CHI*, 2011.

[23] D. P. Kormann and A. D. Rubin. Risks of the passport single signon protocol. *Comput. Networks*, 33(1-6):51–58, 2000.

[24] R. Morris and K. Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979.

[25] R. A. Popa, J. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proc. of USENIX ATC*, 2011.

[26] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proc. of USENIX Security Symposium*, 2008.

[27] Rachna Dhamija and J.D.Tygar and Marti Hearst. Why phishing works. In *Proc. of CHI*, 2006.

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

[29] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *Proc. of ACSAC*, 2009.

[30] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proc. of CCS*, 2009.

[31] S.-T. Sun, Y. Boshmaf, K. Hawkey, and K. Beznosov. A billion keys, but few locks: the crisis of web single sign-on. In *Proc. of NSPW*, pages 61–72, 2010.

[32] J. Thorpe and P. van Oorschot. Human-seeded attacks and exploiting hot-spots in graphical passwords. In *Proc. of USENIX Security Symposium*, 2007.

[33] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proc. of NDSS*, 2006.

[34] M. Wu, R. C. Miller, and G. Little. Web wallet: preventing phishing attacks by revealing user intentions. In *Proc. of SOUPS*, pages 102–113, 2006.

[35] T. Wu. The secure remote password protocol. In *Proc. of NDSS*, 1998.

[36] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2(5):25–31, 2004.

[37] J. J. Yan. A note on proactive password checking. In *Proc. of NSPW*, pages 127–135, 2001.

[38] K.-P. Yee and K. Sitaker. Passpet: convenient password management and phishing protection. In *Proc. of SOUPS*, pages 32–43, 2006.

[39] C. Yue. Preventing the Revealing of Online Passwords to Inappropriate Websites with LoginInspector. In *Proc. of USENIX LISA*, 2012.

[40] C. Yue and H. Wang. BogusBiter: A Transparent Protection Against Phishing Attacks. *ACM Transactions on Internet Technology*, 10(2):1–31, 2010.

[41] Firefox Sync Service. `https://wiki.mozilla.org/Services/Sync`.