# Mercury: Recovering Forgotten Passwords Using Personal Devices*

Mohammad Mannan[1] David Barrera[2] Carson D. Brown[2]
David Lie[1] Paul C. van Oorschot[2]

[1] Dept. of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
[2] School of Computer Science
Carleton University, Ottawa, Canada

**Abstract.** Instead of allowing the recovery of original passwords, forgotten passwords are often reset using online mechanisms such as password verification questions (PVQ methods) and password reset links in email. These mechanisms are generally weak, exploitable, and force users to choose new passwords. Emailing the original password exposes the password to third parties. To address these issues, and to allow forgotten passwords to be securely restored, we present a scheme called *Mercury*. Its primary mode employs user-level public keys and a personal mobile device (PMD) such as a smart-phone, netbook, or tablet. A user generates a key pair on her PMD; the private key remains on the PMD and the public key is shared with different sites (e.g., during account setup). For password recovery, the site sends the (public key)-encrypted password to the user's pre-registered email address, or displays the encrypted password on a webpage, e.g., as a barcode. The encrypted password is then decrypted using the PMD and revealed to the user. A prototype implementation of Mercury is available as an Android application.

## 1 Introduction and Motivation

Users often forget their login passwords. This is not news to the security research community (see [26], a 1993 user study). Forgetting a password leads to a password reset through systems such as Personal Verification Questions (PVQs) or sending the new password to a pre-registered email address. Today's Web has brought about an increase in number of password-protected sites (on average 25 accounts per user [7]), leading to more forgotten passwords. Some users choose weak passwords and reuse them across many sites to cope. Password mangers have mushroomed in most desktop and smart-phone platforms as an offered solution. Some users instead keep notes, written on a piece of paper or stored in a digital file. Others rely on online reset mechanisms offered by most password-protected websites.

* Version: April 5, 2011. Post-proceedings of Financial Cryptography and Data Security 2011. Contact author: m.mannan@utoronto.ca

Weaknesses in online reset techniques pose a significant threat to password based schemes. Several studies—old and new (e.g., [19], [26])—have repeatedly identified these drawbacks: answers to reset questions are publicly available or easily guessed, and even more so by close contacts. Although users may initiate reset requests only occasionally, attackers can misuse these techniques at any time. Indeed, in several real-world incidents, PVQs have been exploited to compromise user accounts [3], [23]. Such techniques have also been exploited by social engineering attacks [15, p. 272]. Beyond security, forgotten passwords often incur significant cost to current IT-dependent organizations. According to Gartner, $20 - 50\%$ of all help desk calls are for password resets, each costing about $70 depending on the organization type.[3]

Despite the critical problem of password loss by large numbers of users, little research effort has been directed towards solutions, compared to online authentication. Several mechanisms (e.g., PVQs and their proposed variants [6], [10], [25]) mandate recalling multiple infrequently-used, text-based secrets to reset a password. The "crime" of password forgetfulness is punished by burdening users with additional cognitive loads. This is clearly not a winning strategy. Others have begun to explore ways to improve such reset techniques, e.g., using pictures (rather than text questions in normal PVQ schemes) as visual cues to prompt for answers [17].

This paper offers a new approach, based on securely restoring existing passwords rather than resetting to new ones. A recent study [24] has confirmed the suspicion that users evolve passwords in a predictable manner, when they are forced to update their passwords. Our key insight is that a secure personal mobile device (PMD), such as a smart-phone or tablet, in combination with a simple to use channel between a PC and the device can solve the difficult problem of *password recovery* (i.e., delivering the original password back to the user) and reduce the need for *password reset* (i.e., creating a new password). Thus, we propose a password recovery mechanism called *Mercury*.[4] The basic idea is as follows: a user creates a public/private encryption key pair on her PMD using entropy from a private object (e.g., an unshared personal image, cf. [13]), or a random source. The public key is shared (e.g., during registration) with the sites hosting the user's accounts. For password recovery, a public-key encrypted password is sent to a registered email address, or displayed on the site's page as a barcode. Using her PMD, the user decrypts the password which is then displayed on the PMD screen. The same public key can be used with different sites. Mercury's focus is to restore forgotten passwords, but it can also be used for traditional password resets, e.g., by sending a (new, system-generated) reset password through Mercury. This scheme can also be used offline for local password recovery, which requires no participation from sites.

[3] See `http://www.mandylionlabs.com/PRCCalc/PRCCalc.htm`. Estimates vary.

[4] In Roman/Greek mythology, Mercury is a messenger of gods and a deity of merchandise and merchants, which we find to be suitable in the context of the proposed recovery method, as passwords are securely delivered from the server to the end-user.

## 2 New Approach: Mercury

### 2.1 Components and Threat Model

**Components.** Main components of the Mercury scheme are as follows: **(*a*)** User PC: The system that will be used to initiate password recovery. It must be capable of establishing a communication channel between itself and a personal mobile device (PMD) in close proximity. The system will relay the encrypted password received from the remote server to the PMD. **(*b*)** Server: The system that stores and will send a user's encrypted password back to her upon request. **(*c*)** PMD: A device capable of performing basic encryption/decryption of data, and capable of establishing a communication channel to the user PC. We use a smart-phone as a PMD. **(*d*)** Mercury software: Application that is used to (re)generate, and store cryptographic keys, transmit public keys to servers and decrypt data received from the user PC. We have implemented Mercury as an Android application. **(*e*)** Local communication channel: The channel used for transmitting the encrypted password from user PC to PMD. Candidate channels include Bluetooth, 2D barcodes such as QR codes,[5] or a USB cable. **(*f*)** Secure offline storage: A secure (physical/digital) location for storing user data necessary for re-generating cryptographic keys in case of PMD loss or upgrade.

| | |
|---|---|
| $U, S$ | User and the website, respectively. |
| $ID_U$ | User ID of $U$ at $S$ (unique in $S$'s domain). |
| $P, A_U$ | $U$'s password and email address as shared with $S$. |
| $M$ | A key generating object selected by $U$. |
| $pubU, privU$ | $U$'s public and private keys respectively, as generated from $M$. |
| $\{data\}_{E_{pubU}}$ | Asymmetric (public-key) encryption of *data* using $pubU$. |
| $PkGen(\cdot)$ | A custom function to generate ($pubU$, $privU$) using a specified source of randomness. |
| $MGen(\cdot)$ | A function to convert data into Mercury-encoded text. |
| $MRet(\cdot)$ | A function to recover data from Mercury-encoded text. |

Table 1: Notation

**Operational assumptions and threat model.** Assume that user $U$ has a PMD running Mercury software (see Table 1 for notation). $U$ generates a public-private key pair (encryption-only). Integrity of the public key $pubU$ is important; $pubU$ must be sent to website $S$ (from the phone or PC) via a secure channel, such as HTTPS. $U$ may verify $pubU$ as received by $S$ as follows: $S$ encrypts a known object (e.g., the site URL, favicon, or site logo) using $pubU$, and transmits it to the user PC. $U$ can then use her PMD to decrypt/verify the known object.

Like any password-only system, we assume that the memorized/recovered password is used only on a trustworthy machine, and the password (hash or plaintext) database on $S$ remains beyond attackers' reach. Attackers may compromise $U$'s recovery email account and initiate a recovery. Email traffic is also

[5] A QR (Quick Response) code is a two-dimensional barcode, see developed in 1994 by a Japanese company called Denso-Wave; see `http://qrcode.com/index-e.html`.

available to attackers, and Mercury does not require email content be encrypted for password confidentiality.

### 2.2 Setup and Recovery Operation

**Account setup.** Assume that $U$ has generated a public and private key-pair for Mercury; see Section 2.3 for key generation and backup. During account creation using a PC browser, $U$ uploads a key file containing *pubU* to $S$. $S$ may display the URL to upload *pubU* as a QR code on the browser; $U$ then scans that URL and the phone forwards *pubU* to the URL. A recovery email address (as customary to many existing systems) $A_U$ is also specified. As an additional step, $S$ may verify both *pubU* (i.e., whether $U$ can prove the possession of *privU* through challenge-response) and $A_U$ (i.e., whether $U$ can access email sent to $A_U$).

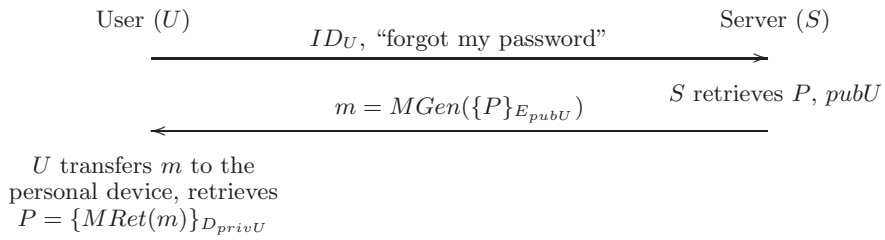


Fig. 1: Password recovery steps

**Password recovery steps.** See also Fig. 1.

1. $U$ sends her user ID $ID_U$ and an indication to recover her password to $S$.
2. Using $ID_U$ as an index, $S$ retrieves $U$'s real password (or optionally assigns her a system-generated temporary password) $P$, and encrypts $P$ using $U$'s pre-shared public key *pubU*. The encrypted result is then converted to a Mercury-encoded format and sent to $A_U$ (or displayed on the PC browser). $S$ notifies $U$ to check her email.
3. $U$ accesses her email and retrieves the Mercury data. The Mercury data is transferred to the PMD via the available communication channel (see Section 2). The device is now used to decrypt $P$ (using the stored/re-generated *privU*). $P$ is then displayed on the device screen.

Note that $U$ can use the same public key for all her services; she is not required to generate a new key-pair for each site. The key-pair is used only for encryption—no signature nor ID management are required (i.e., no need for a revocation list, or other components of a regular PKI).

### 2.3 Key Generation and Backup

We discuss here two methods for user-level public key generation, and key backup procedures. The mobile nature of today's smart-phones and other PMDs makes them more prone to loss. Losing the key pair stored on the personal device would be comparable to losing the master password in a password manager: passwords would no longer be decryptable; see also Section 3: item 3 under "Limitations." To address this issue, we have added a second-level (i.e., only used when the PMD is lost or upgraded) backup that allows the regeneration of the original key pair. Our generation and backup procedures include:

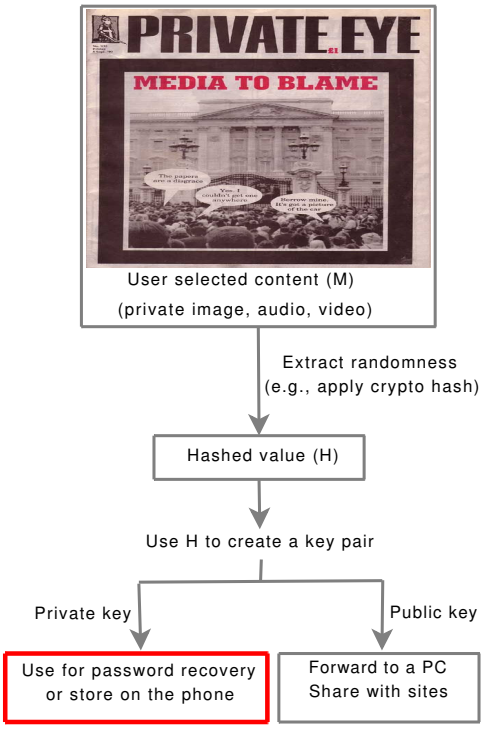


Fig. 2: Mercury key generation from an unshared personal file

**(a) Generate keys using entropy from a private digital object.** Mercury allows $U$ to generate a key pair by seeding the key generator using entropy from a private object. Steps include (see also Fig. 2):

1. $U$ selects a private object $M$ (e.g., personal image, self-composed music) from her phone to be used for the recovery key pair generation.
2. Using a cryptographic hash function $h$, the Mercury program on the phone extracts randomness from $M$: $H = h(M)$. $M$ may be truncated at some value (e.g., 1,000,000 bytes) for efficiency reasons. To gather sufficient entropy, a minimum length for $M$ (e.g., 100,000 bytes) must be enforced.

3. The key pair is generated: $(pubU, privU) = PkGen(H)$. $pubU$ is forwarded to the user PC or remote site $S$, but $privU$ does not leave the phone; in fact, $privU$ may be generated on-the-fly when needed ($PkGen$ ensures that the same key pair is generated from the same $H$ value, i.e., as long as $U$ selects the same $M$).

Backing up a user-selected file appears to be easier than storing keys generated from random sources. It is important to note that this method allows on-the-fly generation of keys so $privU$ does not have to be kept on the device. The user-chosen file must also be stored offline and remain unmodified as even a single bit change may generate different keys. Using the first several hundred bytes may be avoided as some file formats store meta-data (e.g., camera-related information in an image file) in the beginning of a file; meta-data may be updated, e.g., via an image processing program. To ensure sufficient entropy for key generation, low-entropy objects must be avoided, e.g., a single-color image with all pixels having the same value. Users must also refrain from sharing $M$ (e.g., on Flickr or Facebook) as such sharing may compromise Mercury.

**(b) Using a random seed.** Alternatively, Mercury can use a standard key generator using a random seed (i.e., without user-chosen files) and display a string to the user which contains the seed used to generate the keys (e.g., 0xB5CE69ECFFA21082430E6). If the string is given as input to the key generator, it will use that string as a seed, resulting in the same key pair being generated. The random seed in this case (as well for the object-based generation) may be manually written down or converted into a QR code, and the printed copy may serve as a physical backup.

### 2.4 Variants

**Mercury with symmetric keys only.** A symmetric-key only variant of Mercury may be called *password recovery through reverse cookies*; instead of having the server store a cookie on a client machine (as customary to the use of browser cookies), it is the client that stores a password cookie on the login server. The server sends the cookie to the client when the client requests help remembering a password.

In this mode, $U$ generates a symmetric key ($K = K_U$) with one of the methods of Section 2.3, but instead of sending the key to $S$ during account creation, $U$ sends an encrypted string of her password ($\{P\}_K$) for the website to store (along with her email address and hashed password as usual). For password recovery, $S$ sends $\{P\}_K$ to $U$'s email, where she can use a PMD with Mercury to decode and decrypt her password using $K$. In this mode, $K$ can also be generated on-the-fly from $M$ during password recovery, or stored on the PMD. Since the same key is used for encryption and decryption, $K$ must be kept secret.

The downside of using the symmetric key variant of Mercury is that when the user wants to update her password, a new encrypted password must be sent in parallel. Advantages of using the public key version of Mercury over this variation are discussed in Section 3.

**Alternative data channels.** While Mercury can be implemented using any type of data channel between the PC and the personal device, some channel properties are desirable. The channel should allow simple (minimal setup or need for additional software) and fast communication between devices. QR codes are a good candidate since they transfer data by simply taking a picture with the user device. The use of QR code is rapidly increasing, and is apparently more engaging for users than other channels. However, all cell-phones may not feature a camera.

Universally available audio channels on cell-phones may also be used as an alternative. $S$ can send an encrypted password encoded as an audio file; $U$'s PC can play the tune and the cell-phone application can decode and decrypt the password (cf. Shazam[6]). As mentioned previously, Bluetooth or USB can also be used as a data channel, but require additional setup such as device pairing and carrying cables which may reduce convenience.

**Variation with PMD as primary device.** If a PMD is used as a primary email communication device (which is already common for many users) and the recovery email is accessed directly from the PMD, then neither the PC nor the PC-to-PMD channel in the basic Mercury protocol is required. Here text messages (SMS) may also be used, instead of email, to send the encrypted password directly to a smart-phone.

**Variation without a PMD.** We have also considered a version of Mercury which operates without a PMD by generating and storing keys on the PC exclusively. To avoid the loss of portability and the requirement of storing long-term keys on the PC, keys or key-generating objects may be stored on mobile storage, e.g., USB flash drives, albeit at the risk of introducing different problems (e.g., lost/stolen drives).

**Storing passwords: hashed, cleartext, or encrypted.** Mercury is transparent to how site maintainers store user passwords (cleartext, hashed, or encrypted). Security proponents recommend that only hashes of user passwords be stored to prevent direct (e.g., without running a dictionary attack) reuse of passwords in case the (hashed) password list is compromised. Password recovery requires access to the original cleartext password, but Mercury can also work without access to it. For example, encrypted temporary passwords may be sent to PMD, recovered and used to reset a password; see also the symmetric-key variant of Mercury in Section 2.4. Alternatively, a site can store the hashed password, user public key, and public-key encrypted password (instead of storing the cleartext password). The encrypted password is then forwarded when requested. Note that password update in this case is affected by whether the public-key encryption is done on the server side or at the client browser. If encryption is performed at the server side (i.e., the site receives the initial cleartext password), during password update, the site can replace the old encrypted password with the updated one, without requiring any extra step from the user. For browser-

[6] A music discovery tool for cell-phones, see: `http://www.shazam.com/`. We acknowledge N. Asokan for proposing the use of audio channel in Mercury.

side encryption (i.e., the site never receives cleartext password), the user must have the public key available during the password update.

## 3 Features and Limitations

In this section, we list several features and limitations of Mercury.

**Advantages and features.**

1. There is no need to trust third-parties in Mercury. Just as in the current practice, $P$ is shared only between $U$ and $S$.
2. If an encrypted password is compromised, e.g., by a malicious ISP or email provider, attackers cannot run an offline dictionary attack as $P$ is encrypted with $pubU$ (for RSA, using the EME-OAEP padding as defined in PKCS #1 v2.1 [11]).
3. Updating a password remains the same as current strategies; users resort to Mercury only when a password recovery is needed, and in the public-key variant do not require any extra step to "sync" their updated passwords (as opposed to traditional password managers).
4. Users can easily replace their personal device with a new one. However, the key-generating file must be deleted from the old device and copied to the new one; this step is simpler when the file is stored on a removable memory card. Also, the Mercury program must be installed on the new device. Note that the key regeneration technique using a user-chosen private object may simplify several user-level key management issues such as key transfer and backup. Such public keys without a PKI might enable easier adoption of other security applications including PGP, and authentication (cf. [22], [12]).
5. Mercury may be used for password recovery in a local PC. Some consumer systems, e.g., Mac OS X, Ubuntu Linux, offer users to setup a "password hint" (a text string that may help $U$ to recall her login password). If $pubU$ is stored on the local PC, then the login procedure may be modified to enable Mercury as follows. When $U$ indicates that she lost her password, a temporary password is encrypted with $pubU$, and displayed as a QR code. $U$ can use her smart-phone to retrieve the password. Note that most current operating systems store only password hashes, thus requiring the use of temporary password in this case. Alternatively, the original password may be stored encrypted under $pubU$.
6. Once the public key $pubU$ is stored on $S$, $S$ can securely communicate additional information to $U$. For instance, a banking site could enable two-factor authentication by requesting the user's password, and sending a one-time password encrypted with $pubU$. $U$ decodes the one-time password and submits it back to $S$, proving that she is in possession of the corresponding private key $privU$.

**Limitations.**

1. Mercury requires service providers' assistance for online deployment. However, some sites may be reluctant to implement Mercury as not all users

possess an additional personal device, or would be willing to use this service. To deal with such cases, Mercury can be deployed gradually, e.g., as an option so that users who want to take advantage of it, can do so without imposing any requirements on others.
2. Users must carry their device for portability (i.e., password recovery from anywhere). However, users can continue regular login without Mercury.
3. To deal with a lost or stolen personal device, users must keep a "secure" backup of the personal object or random seed used for the key generation; e.g., storing the object on a USB key and keeping it private, see Section 2.3. If there is no backup, $U$ must generate a new key-pair, login to all Mercury-enabled sites (using recalled passwords), and then upload the new public key. Note that losing the keys or not having access to the PMD do not necessarily restrict login, as long as $U$ has access to her primary login password.
4. Users may willingly or accidentally upload the key-generating private object to public sites such as Facebook. Making users understand the risk of such actions appears non-trivial, especially for users who want to share all their digital content with both friends and strangers. However, this risk is reduced by using a random seed for generating keys and storing the seed offline, e.g., as a printed QR code (see Section 2.3, item (b)).

**Smart-phone compromise.** If the smart-phone or other personal device is infected with malware, the user's private key might become available to an attacker. However, the key alone may not be enough to gain access to user accounts. The attacker needs access to both the key and the encrypted password sent during password recovery. Recall that the password is sent to $U$'s email address, which would require an attacker to also obtain the email account password. If the recovery email is accessible from the compromised phone (e.g., logged in email client on the phone), attackers may be able to compromise passwords for all Mercury-protected accounts of a user. Operating system security mechanisms such as built-in secure storage facilities and file system isolation might help lower the risk of this attack. For the case of physical device theft, we recommend the use of a PIN-lock[7] on the personal device.

## 4 Prototype Implementation on Android

Mercury has been implemented as a proof of concept application for Android. Virtually all Android phones are equipped with cameras, network connectivity, and libraries for performing the basic functions of Mercury (generating key pairs, RSA public key encryption/decryption, and data transmission over HTTP/HTTPS). Thus, we use QR Codes as the data channel to transmit encrypted passwords from the user PC to the smart-phone. The prototype consists of two parts: the client, an Android application; and the server, a demo of a normal usage scenario for Mercury. The Android application is also capable of

[7] Apple iOS and other smart-phone platforms allow the user to specify a number of password entry attempts after which to format the device memory.

operating in a standalone way without a server-side component, enabling users to immediately benefit from Mercury. We describe the implementation details below, for an Android client.

**Key pair creation.** The Bouncy Castle Crypto API (`bouncycastle.org`) is used to generate a 1024-bit RSA key pair $(pubU, privU)$. Keys are generated using Java `SecureRandom` as the source of entropy, and stored with the PKCS#1 Optimal Asymmetric Encryption Padding (OAEP) option [11]. OAEP helps prevent guessing attacks on the encrypted string. The key pair is stored on the device using the Android data storage, saving files into a dedicated per application directory [2]. By default, this directory is accessible only to the given application (enforced through standard UNIX file permissions, as each application runs with a different user ID). Upon launch of the application for the first time, users are instructed to generate a key pair from the application menu.

**Generating an encrypted password.** Once a key pair has been created, $U$ can proceed to either create an account on a web server that supports Mercury password recovery, or create new Mercury encrypted password for personal use. In the latter case, $U$ selects the "Encrypt password" menu option (Fig. 3a), and inputs a password $P$. $P$ is then encrypted with $pubU$, and the result is encoded to Base64 before being converted to a QR code ($qr = MGen(\{P\}_{E_{pubU}})$). Base64 encoding is necessary to create a valid QR code of ASCII characters. The application can email $qr$ to $U$, which can be printed or stored for future use.

**Scanning a password for recovery.** A QR code of the encrypted password is scanned using the ZXing[8] ("Zebra Crossing") barcode reader library for Android. If ZXing is not installed, the Mercury app prompts $U$ and opens the Android Market page for the Barcode Reader application that provides the ZXing service. The scanned QR code will be decoded from Base64 and then decrypted so that $\{MRet(qr)\}_{D_{privU}} = P$; see Fig. 3.


(a) Mercury startup (b) Password recovery on Mercury-enabled site (c) Decrypted password

Fig. 3: User interface for password recovery

[8] `http://code.google.com/p/zxing`

**Transmitting public keys to a server.** We have implemented a feature to transmit the generated public key directly from the phone to a server. This feature works by having the user scan a QR code containing the URL of the server and a nonce. The Mercury application then generates a POST request to the scanned URL that includes the public key $pubU$. The key is saved into $U$'s account on the server.

**Web server.** Our prototype server is a typical "Signup/Login/Forgot password" system using PHP and MySQL. $U$ creates a new account; the server $S$ stores $(ID_U, P)$, and then prompts $U$ to send her public key. $S$ verifies that the nonce is valid, and stores $pubU$ in a database.

In a typical use scenario, $U$ attempts to log in, and realizes she has forgotten $P$. Clicking the "forgot password" link after specifying $ID_U$ emails her a QR code generated by encrypting $P$ with $pubU$. Alternatively the QR code can be displayed immediately on the screen. $U$ then uses her phone to scan the QR code and retrieve $P$.

**Limitations of the prototype.** Our prototype does not use a secure application data storage, such as the Android Credential Storage, available in versions 1.6 and higher. Despite this, the storage we used has `UID/GID` protection mechanisms offered by all versions of Android, which prevents the private key from being accessed by other applications. Future versions of Mercury will make use of the secure storage facility to reduce the risk of private key leaks.

Mercury currently uses 1024-bit RSA keys and the key length is hardcoded in this prototype. We have tested Mercury on Nexus One phones, and operations such as key generation, public key sharing and password recovery were completed without any noticeable delay from a user's perspective. During testing, we found that 2048-bit keys add a small but non-perceptible delay to Mercury operations on the Nexus One. Older devices with slower processors may exhibit a more noticeable delay with larger key sizes.

## 5 Existing Approaches, Related Work and Comparison

### 5.1 Existing Password Recovery Approaches

Below we discuss common techniques used for password recovery and managing multiple passwords.

**Password managers.** There are two types of common password managers: local and remote/third-party-supported. In the first case, passwords (user-chosen or system generated) are stored on the user's PC (as a file managed by the password manager application, or integrated with a web browser), or on a PMD. It is generally recommended that the password file must be encrypted with a key derived from (or protected with) a password, called the master password. However, most users do not use such passwords, as with the built-in browser password managers [20]. The master password is susceptible to offline attacks if a copy of the encrypted password file can be obtained. From a usability perspective, several issues may arise: losing the master password may result in losing

access to many accounts; the local password file must be backed up; the file must also be carried with the user for portability (unless it is already on a PMD); and if a password is updated at a site, the user must also update it in the manager.

When the password file is stored online with a third-party, users must trust the service provider to be honest, even if the password file is sent encrypted from a user machine. The remote party can be actively malicious, or become compromised; strong incentives exist for attackers as such a compromise may enable access to many accounts per user.

**Email password recovery/reset.** An email address is registered at the time of account creation; when a user provides her user ID and indicates that she has lost the account password, the registered email is used for sending the original password, or a temporary password/URL to reset the account password. As long as users can access the email account, this solution is portable. It is also quite instant in today's Internet, as email delivery is close to real-time. Because of low-cost deployment, and the ubiquitous use of email, email-based recovery/reset is most commonly used. One password survey [4] reports that about 92% of sampled websites offer an email-based solution, with 44% sending a reset link in the email, 32% sending a temporary (reset) password, and 24% sending the original cleartext password.

The unencrypted nature of email enables the email provider (or someone compromising the provider) to learn all reset links and passwords. Any intermediate party (e.g., ISPs, wireless access points) between the user and email server can also access the reset emails when sent via an unencrypted channel such as regular HTTP webmail as provided by Yahoo and Hotmail sites; optional HTTPS-protected sites are also not always immune to attacks, see [9]. Other pitfalls include [8]: recovery emails being classified as spam, and an exposed cleartext passwords may be exploited at other sites due to password reuse.[9] As email is being used for increasingly important authentication tasks (e.g., domain registration and management), compromised accounts can result in costly consequences [1].

**Other methods.** Preregistered questions/answers are sometimes used to reset an account password. Concerns include: (i) users may forget answers to these questions; and (ii) these answers may be easier to guess than a user-chosen password (see e.g., [16], [19]). Few other solutions are deployed in practice. Google offers sending a reset code as an SMS to a pre-registered cell-phone number. EBay allows three methods for reset: answering personal profile information (e.g., postal code, phone number, and date of birth), sending reset code via an automated call-back to a phone number, and instant messaging based live help on the eBay site.

[9] One survey [21] reported that 73% of online banking users share their banking password to access at least another less sensitive account.

### 5.2 Comparison: Mercury vs. Current Approaches

Table 2 compares Mercury to other password recovery/reset approaches. For the feature comparison and analysis listed below, we assume Mercury's primary mode of operation based on a private-public key pair.

| | Independence of $3^{rd}$ party trust | Portable | Recovers original password | Overhead for account creation | Transparent to password updates |
|---|---|---|---|---|---|
| Pass. managers (online) | No | Yes | Yes | High | No |
| Pass. managers (offline) | Yes | No | Yes | High | No |
| Email | No | Yes | No[10] | Low | N/A |
| PVQs | Yes | Yes | No | High | N/A |
| Mercury | Yes | Yes | Yes | Low | Yes |

Table 2: Comparison of Mercury to other password recovery/reset methods.

**Independence of $3^{rd}$ party trust.** Online password managers and email require the user to trust the transmission channel between the server and the PC, as well as ISPs and email providers. Offline password managers require no third-party trust, but the user must manually keep track of all accounts in a password file. Mercury does not require trust in third-party sites, communication channels, or service providers.

**Portability.** Mercury offers portability if the user is in possession of the personal device at the time of password recovery. Other recovery/reset systems listed have some degree of portability, but may require the user to carry a password file at all times (e.g., offline password managers). This may reduce portability and increase inconvenience for the user.

**Restoring original password.** PVQs do not typically allow the user to obtain the original password, but rather allow the user to set a new password after successfully answering verification questions. Email based recovery/reset systems frequently send the user a link for password reset, or a one-time temporary password. Password managers, as well as Mercury allow the user to retrieve the original password.

**Overhead for account creation.** Password managers and PVQs must be updated or configured every time a new account is created. Overhead is small if there is an automatic way to keep track of new passwords (e.g., browser-based password managers). Mercury requires the user to send the public key to the server, also adding an extra step to account creation; this step can be implemented (e.g., as in our Android prototype) to have minimal time overhead.

**Transparency to password updates.** When a password is updated, the corresponding account in a password manager must be updated as well. Similar to the account creation phase, some password managers might detect password updates and store them automatically. Mercury, by design, offers full transparency

[10] Most sites email reset links/passwords as opposed to original passwords [4].

to password updates in the public key mode, allowing users to update their passwords as usual without additional steps.

### 5.3 Other Related Work

Here we discuss other related proposals involving user-level public keys and online authentication. Existing techniques for password recovery are discussed in Section 5.1.

Seeing is Believing (SiB) [14] uses QR codes to transfer a public key from one user device to another (equipped with a camera) as part of authenticated key establishment between devices with no previously shared context. The SiB proposal is used mainly for authenticating two devices in close proximity. Mercury relies on an existing authentication channel, and is used only when credentials are forgotten.

Snap2Pass [5] uses QR codes and a camera phone to implement a challenge-response protocol for authentication; this is envisioned as a replacement for password-based authentication. In contrast to replacing passwords, which are currently deeply entrenched into the web ecosystem, the focus of Mercury is to facilitate and simplify secure password recovery.

Some password managers, e.g., PwdHash [18] generate site-specific passwords from a single, user-chosen password. If adopted widely, these may help reduce the need for password reset as users are expected to remember only one master password. However, the master password is vulnerable to offline dictionary attack, if a site-specific password and the site URL is known (e.g., in a phishing attack). Forgetting the master password results in denied access to all PwdHash-protected accounts; access may be regained via resetting all such passwords.

Mercury does not aim to replace passwords as used. It is targeted towards password recovery, and as such, it is designed for infrequent use. It employs user-level public keys, which can be reconstructed from user-chosen digital objects. No new private or secret keys are shared between the phone and website.

## 6 Concluding Remarks

Password recovery and reset techniques are as old as passwords themselves. Before the Internet, such techniques were relatively simple and secure; e.g., users could talk to an administrator in person and reset their passwords. In current large-scale enterprise environments, and online sites with millions of users, existing password recovery techniques are inadequate and costly in terms of security, usability and expense. The proposed online password recovery technique – Mercury – avoids limitations of current password managers (both online and offline), email reset methods and PVQs.

In contrast to traditional approaches which require users to create a new password when the old one is forgotten, Mercury allows what may seem like a small, but we believe important paradigm shift: recovering an existing password in a secure manner. Mercury's security is independent of plaintext email

or the compromise of a "master password" in password managers. Mercury is also transparent to the server-side password storage method, as well as to the communication channel between the PC and the personal device. In the current implementation of Mercury, we take advantage of widely available Android smart-phones as a PMD, and increasingly-used 2D barcodes as an engaging communication channel between the PC and the phone. The Mercury Android application and a demonstration site are available for public testing at: `http://www.ccsl.carleton.ca/software/mercury/`.

**Acknowledgments.** We thank N. Asokan and anonymous referees for their helpful comments. The first author is supported by an NSERC PDF. The last author acknowledges NSERC funding under a Discovery Grant and as Canada Research Chair in Authentication and Computer Security. Partial funding from NSERC ISSNet is also acknowledged.

## References

1. D. Ahmad. The confused deputy and the domain hijacker. *IEEE Security an Privacy*, 6(1), 2008.
2. Android Open Source Project. Data storage (android developers). `http://developer.android.com/guide/topics/data/data-storage.html`.
3. BBCNews.com. Obama Twitter account 'hacked by Frenchman'. Mar. 24, 2010. `http://news.bbc.co.uk/2/hi/8586269.stm`.
4. J. Bonneau and S. Preibusch. The password thicket: Technical and market failures in human authentication on the web. In *Workshop on the Economics of Information Security (WEIS'10)*, Cambridge, MA, USA, June 2010.
5. B. Dodson, D. Sengupta, D. Boneh, and M. S. Lam. Secure, consumer-friendly web authentication and payments with a phone. In *Conference on Mobile Computing, Applications, and Services (MobiCASE'10)*, Santa Clara, CA, USA, Oct. 2010.
6. C. M. Ellison, C. Hall, R. Milbert, and B. Schneier. Protecting secret keys with personal entropy. *Future Generation Computer Systems*, 16(4), Feb. 2000.
7. D. Florêncio, C. Herley, and B. Coskun. Do strong web passwords accomplish anything? In *USENIX Workshop on Hot Topics in Security (HotSec'07)*, Boston, MA, USA, Aug. 2007.
8. S. Garfinkel. Email-based identification and authentication: An alternative to PKI? *IEEE Security and Privacy*, 1(6), 2004.
9. Guardian.co.uk. Gmail ups security after Chinese attack. News article (Jan. 13, 2010). `http://www.guardian.co.uk/technology/2010/jan/13/gmail-increases-security-chinese-attack`.
10. M. Jakobsson, E. Stolterman, S. Wetzel, and L. Yang. Love and authentication. In *Conference on Human Factors in Computing Systems (CHI'08)*, Florence, Italy, Apr. 2008.
11. J. Jonsson and B. Kaliski. Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1. RFC 3447 (Feb. 2003). Category: Informational.
12. J. Lopez, R. Oppliger, and G. Pernul. Why have public key infrastructures failed so far. *Internet Research*, 15(5), 2005.

13. M. Mannan and P. van Oorschot. Digital objects as passwords. In *USENIX Workshop on Hot Topics in Security (HotSec'08)*, San Jose, CA, USA, July 2008.
14. J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentiction. *Security and Networks*, 4(1–2), 2009.
15. K. Mitnick and W. L. Simon. *The Art of Deception*. Wiley, 2002.
16. A. Rabkin. Personal knowledge questions for fallback authentication. In *Symposium on Usable Privacy and Security (SOUPS'08)*, Pittsburgh, USA, July 2008.
17. K. Renaud and M. Just. Pictures or questions? Examining user responses to association-based authentication. In *British HCI Conference*, Dundee, Scotland, UK, Sept. 2010.
18. B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security Symposium*, Baltimore, MD, USA, 2005.
19. S. Schechter, A. J. B. Brush, and S. Egelman. It's no secret. Measuring the security and reliability of authentication via 'secret' questions. In *IEEE Symposium on Security and Privacy*, May 2009.
20. B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *ACM Computer and Communications Security (CCS'09)*, Chicago, IL, USA, Nov. 2009.
21. Trusteer.com. Reused login credentials. Security advisory (Feb. 2, 2010). `http://www.trusteer.com/sites/default/files/cross-logins-advisory.pdf`.
22. A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium*, Washington, D.C, USA, 1999.
23. Wired.com. Palin e-mail hacker says it was easy. Sept. 18, 2008. `http://www.wired.com/threatlevel/2008/09/palin-e-mail-ha/`.
24. Y. Zhang, F. Monrose, and M. Reiter. The security of modern password expiration: An algorithmic framework and empirical analysis. In *ACM Computer and Communications Security (CCS'10)*, Chicago, IL, USA, Oct. 2010.
25. M. Zviran and W. J. Haga. Cognitive passwords: The key to easy access control. *Computers & Security*, 9(8), 1990.
26. M. Zviran and W. J. Haga. A comparison of password techniques for multilevel authentication mechanisms. *Computer Journal*, 36(3), 1993.