TruWalletM: Secure Web Authentication on Mobile Platforms

Sven Bugiel[1], Alexandra Dmitrienko[2⋆], Kari Kostiainen[3],
Ahmad-Reza Sadeghi[1], Marcel Winandy[2]

[1] Fraunhofer-Institut SIT Darmstadt, Technische Universität Darmstadt, Germany
{sven.bugiel, ahmad.sadeghi}@cased.de
[2] System Security Lab, Ruhr-University Bochum, Germany
{alexandra.dmitrienko, marcel.winandy}@trust.rub.de
[3] Nokia Research Center, Helsinki, Finland
{kari.ti.kostiainen}@nokia.com

Abstract. Mobile phones are increasingly used as general purpose computing devices with permanent Internet connection. This imposes several threats as the phone operating system (OS) is typically derived from desktop counterparts and, hence, inherits the same or similar security shortcomings. In particular, the protection of login credentials when accessing web services becomes crucial under phishing and malware attacks. On the other hand many modern mobile phones provide hardware-supported security mechanisms currently unused by most phone OSs.
In this paper, we show how to use these mechanisms, in particular trusted execution environments, to protect the user's login credentials. We present the design and implementation proposal (based on Nokia N900 mobile platform) of TruWalletM, a wallet-like password manager and authentication agent towards the protection of login credentials on a mobile phone without the need to trust the whole OS software. We preserve compatibility to existing standard web authentication mechanisms.

1 Introduction

Today's smartphones offer compelling computing and storage capacity allowing the realization of various applications such as web browsing, e-mailing, multimedia entertainment, location tracking, and electronic purchase. The mobile web browser on the phone enables users to access standard Internet web sites, including those that require an authentication of the user (typically username/ password). However, the popularity of mobile phones and the vast number of supported applications makes these platforms also more attractive to attackers.

While smartphones are very flexible computing devices, they generally do not provide sufficient security to protect user credentials. As their operating systems are derived from desktop counterparts, they are vulnerable to malware, phishing

⋆ Supported by the Erasmus Mundus External Co-operation Window Programme of the European Union

and physical attacks. While wallet-like authentication agents [1,2,3,4,5] exist that automatically manage mutual authentication between the user's computing device and a remote web server, they have been demonstrated for PCs only, where resource constraints are not vital. Due to the resource constraints of mobile devices, the development of such secure systems is more challenging than for PCs, in particular to provide a secure (graphical) user interface – although promising research prototypes have been demonstrated recently [6].

Currently, the only solutions available for mobile phones to secure user credentials are password managers, which are essentially simple password databases. They encrypt all passwords with a master password but do not offer sophisticated protection against malware or classical phishing attacks.

On the other hand, many modern mobile phones are already equipped with a hardware-supported secure trusted execution environment (TrEE) which allows the secure and isolated execution of certain code, and which can also protect certain amount of data persistently. However, usually the size of the code and data that can be processed in a TrEE is very small, thus wallet-like architectures developed for PCs are not directly suitable. If the authentication protocols are changed and public key-based credentials are used, the TrEE can be used to provide a secure on-board credentials platform (ObC) [7]. However, an ObC-based authentication is not compatible with existing web-based authentication on most servers.

Thus, the challenge we have to face is to provide a reliable and secure protection of login credentials on a mobile phone that can take advantage of hardware-supported security mechanisms while using the standard OS on the phone, and while being compliant to standard web authentication schemes.

Contribution. In this paper, we present $TruWalletM$, a hardware-assisted wallet-based web authentication as a mechanism to protect user credentials from malware, phishing and physical attacks on mobile platforms. We present the design and implementation proposal of our solution, which also meets important usage requirements: (i) compatibility with existing password-based web authentication, (ii) software reuse of legacy OS and legacy web-browser on smartphones, and (iii) not noticeable performance overhead. To the best of our knowledge, no other solution exists for mobile phones that meets these requirements. Our key design solution, which allows us to meet these requirements, is to split a single SSL/TLS connection between the user device and the server into two logically separated channels, where one is protected by TrEE and is used to transmit passwords, and another one is intended for conventional data. Our design relies on the availability of a Trusted Execution Environment (TrEE) to protect the execution of critical code from tampering. Such TrEEs can be provided by commodity secure hardware for mobile devices such as M-Shield [8] and ARM TrustZone [9].

2 Model and Requirement Analysis

In this section we consider the model and the security and functional objectives and requirements that are desired. We then discuss in Section 6 which of them can be achieved by our $TruWalletM$ design and implementation proposal on top of currently available general purpose mobile platforms.

System model and use case. Our system model involves the following parties: (i) a user, (ii) a device (mobile platform), (iii) a web server, and (iv) an adversary. We consider application scenarios where a user deploys his mobile device to access services provided by remote web servers (over the Internet). The access is granted through an authentication protocol where the user authenticates using username/password.

Adversary model. The main goal of the adversary is to obtain unauthorized access to credentials and the services (provided by the web servers) that usually the user has access to. The threats in this context are:

- *Software attacks.* The adversary may inject malicious software into a device, or exploit the vulnerabilities of the existing application (e.g., web browser). This allows the adversary to access user credentials in device memory, read them out from the login form of the browser when they are inserted by the user during login procedure, eavesdrop on the user input interface (e.g., key-loggers) and communication channel with the web-server, invoke credentials usage or launch phishing attacks[4].
- *Password-related attacks.* The adversary may perform dictionary or brute-force attack in order to recover passwords, or the adversary may apply the credentials, he has learned from an attacked web-server, to other web-services of the user, as many users tend to reuse credentials for different web-sites.
- *Physical Attacks.* The adversary may obtain physical access to the device (e.g., by stealing the device or accessing it while it is left unattended), invoke the authentication procedure on behalf of the user, or tamper with the underlying hardware.

Security objectives. Our security objectives to address these threats are:

- *Protection of user credentials.* User credentials must not be accessible or forged by unauthorized entities (protection of integrity and confidentiality). This requirement ensures that the adversary cannot impersonate the legitimate user.
- *Trusted path.* A secure channel between the user and the web server must be established. This requirement ensures that the user is communicating with a correct web server and that operations are invoked by the legitimate user.

[4] Here the adversary attempts to trick users to reveal their credentials to a server under his control, for instance by luring the user to a faked web-site (classical phishing), or by malware displaying forged login pages (malware phishing).

Functional objectives. In addition to the security aspects, our $TruWalletM$ design should consider important functional objectives that are essential for practical deployment:

- *Software Reuse.* For many mobile platforms installing an alternative OS is not an available option. Also, the browser should be used as is, as installing a patch originating from a third party developer might not be possible[5].
- *Interoperability.* As the majority of web servers rely on password authentication, our solution supports this method. Other authentication methods, such as OAuth, are complementary to our work and can be easily integrated into our architecture.
- *Low performance overhead.* We require imposed performance overhead to be feasible for mobile devices. We require that users should not notice any delay while they are browsing Internet web pages, but we accept more significant delays for short time periods for performing critical operations such as authentication.

Assumptions. We rely on availability of a secure hardware which provides a Trusted Execution Environment (TrEE) with following features: (i) isolated secure code execution, (ii) secure storage, (iii) integrity protection of secure execution environment. Such TrEEs can be provided by general purpose secure hardware such as M-Shield [8] and ARM TrustZone [9]. We assume the TrEE provided by secure hardware is tamper-protected[6].

For secure communication between the device and the remote web server we utilize SSL/TLS protocol. We assume that all cryptographic primitives of SSL/TLS protocol are secure. Also, we rely on a trustworthy SSL/TLS Public Key Infrastructure (SSL-PKI) used (implicitly) to authenticate the server during SSL/TLS channel establishment.

We do not consider denial-of-service (DoS) attacks, since in the context of our adversary model it is impossible to prevent them. In fact, an attacker who has control over the user environment or even has physical access to the device can always cause DoS, e.g., by switching off the device.

We do not consider attacks where an already established server connection is misused by malware (such as transaction generators), but rather concentrate on protection of user credentials. However, our design can be extended with an transaction confirmation agent similar to the solutions in [3,11].

[5] For instance, the Android security architecture would require the patch to be signed with the same developer signing key as the patching application.

[6] It is tamper protected to some degree, e.g., resistant against standard side-channel attacks. However, the severeness of hardware attacks depends on the effort: an example is the recently reported hardware attack on the Trusted Platform Module (TPM) [10].

3 Architecture and Design Decisions

3.1 Design Decisions

It is a challenging task to design an architecture which fulfills requirements defined in Section 2. For instance the requirement to reuse software (particularly OS) rules out virtualization-based approaches to provide runtime isolation of the critical code operating on user credentials. Instead, our design relies on isolated execution of trusted code within the TrEE. However, currently TrEEs provided by commodity secure hardware are very resource constrained, e.g., M-Shield has about 10-20 Kb memory available in the secure mode. On the other hand, a standard user/password authentication protocol assumes transmission of the password to the server through a SSL/TLS connection. In our design this implies running the code handling SSL/TLS communication within the TrEE in order to (i) protect user credentials and (ii) be compatible with legacy web-servers. However, this is not possible due to size of this code. Thus, one of the main challenges we face is to handle SSL/TLS channels with a web server in a secure way as this cannot be done within the TrEE. We overcome these limitations by letting the trusted and untrusted code cooperate in establishing a secure channel between the user platform and the web server.

Handling SSL/TLS connection in collaboration with trusted code running within the TrEE is expensive as it requires multiple switches between normal and secure execution mode[7]. On the other hand, high performance overhead contradicts the requirement to keep performance overhead acceptable. To address this problem, we use two separated logical SSL/TLS subchannels over a single SSL/TLS connection to the server: One is partially handled within TrEE and is used for performing security sensitive operations such as login, registration or password change, while another one does not require invocations of the secure side and is used for transmitting conventional data (i.e., the content of web pages). To manage two logical SSL/TLS subchannels, we utilize the standard SSL/TLS resume mechanism, which is intended to reinstate a previously negotiated SSL/TLS session between a client and a server. SSL/TLS resume mechanism is widely supported, according to statistics provided by [12], 91% of web servers support SSL/TLS resume protocol.

3.2 Architecture

Components. The execution environment of a mobile platform is divided into two isolated parts: "open world" and the TrEE. $TruWalletM$ consists of two main components, WalletHelper (in open world) and WalletCore (in TrEE). The main work is done by WalletHelper (e.g., parsing pages), while WalletCore is only intermittently invoked to perform security-critical operations. "Open world" also contains an operating system (OS), a commodity mobile web browser WebBrowser, a secure storage SecureStorage and a trusted user interface TrustedUI.

[7] A single invocation of secure side requires about 1.9 ms for M-Shield TrEE.

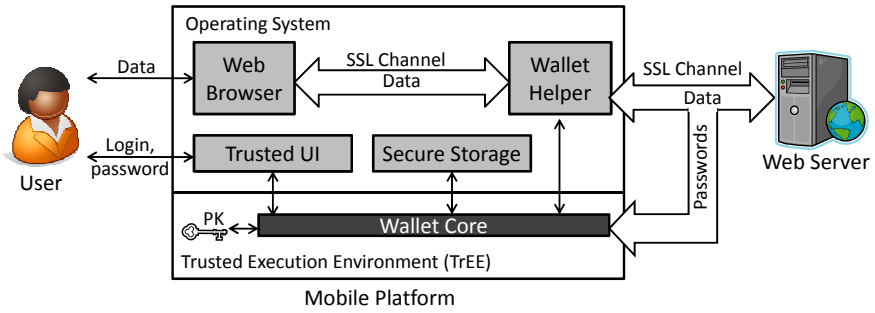

Fig. 1. TruWalletM Architecture

WebBrowser is typically used by the user as a tool to login to remote web servers. SecureStorage is a database where passwords and other security sensitive data are stored persistently and securely. TrustedUI provides trusted user interface between TrEE and the user. The user customizes TrustedUI by adding a unique phrase or a background picture such that he can distinguish TrustedUI from interfaces forged by malware [13,14]. User-specific interface elements are stored within the secure storage and are accessible only by the TrustedUI component.

WalletHelper and WebBrowser are untrusted to handle user credentials, also the OS may run arbitrary software including malware. In contrast, TrustedUI is (partly) trusted, although it is an OS-side component. Ideally, TrustedUI should reside within the TrEE. However, the constraint resources of contemporary trusted execution environments, e.g., 20 kB of secure RAM for M-Shield, prohibit such an implementation at the present time. Thus, in our current architecture TrustedUI is realized as an OS component[8]. However, future generations of the TrEE may provide the required capabilities to implement a fully trusted path between the user and the TrEE, because hardware vendors continue to extend their TrEE's capabilities[9]. For now, trust into the OS components can be provided by ensuring the component's integrity by means of *secure boot.* [10] – a mechanism currently supported by TrEEs. However, secure boot does not protect against runtime attacks, thus trust to this component is limited. In Section 6 we will discuss security implications of runtime compromise.

WalletHelper acts as an SSL/TLS proxy and maintains two SSL/TLS connections, one to a web server WebServer and another one to WebBrowser. SSL/TLS connection to WebBrowser is a regular one and is not protected by additional

[8] Although the availability of such an interface would greatly enhance the security of our architecture, the problem of implementing a TrEE-based (G)UI is orthogonal to the work in this paper.

[9] For example, the next-generation M-Shield increases the size of secure RAM to 40-60 kB.

[10] The boot process is terminated in case the integrity of a component to be loaded could not be verified (e.g., it does not match to a securely stored reference value) [15]

measures. In contrast, SSL/TLS connection with WebServer is established in cooperation with WalletCore and consist of two logical subchannels. WalletCore protects the shared secret with WebServer and controls switching of subchannels. When login, registration or password change protocols are initiated, the logical channel is switched to WalletCore. Before passwords are used, WalletCore ensures they belong to the appropriate web server. When the protocols have been accomplished, the logical channel is switched back to WalletHelper. In this way regular communication does not require invocation of WalletCore and, thus, does not impose additional overhead.

To prevent unauthorized credential usage by other users, the wallet requires user authentication (e.g., a user password) to login into the wallet. The authentication is done through TrustedUI. In this way, passwords stored by the wallet are bound to the corresponding user.

4 Use Cases and Protocols

In this section we describe the following use case scenarios: establishing SSL/TLS connection and managing two logical SSL/TLS subchannels, initialization, registration, login and password change. We also provide corresponding protocols.

4.1 Establishing SSL/TLS connection and managing two logical SSL/TLS subchannels

Figure 2 illustrates the SSL handshake protocol. The client and server exchange $ClientHello$ and $ServerHello$ messages and negotiate the required system parameters. In the following steps the server's certificate $Cert_S$ is sent and verified, the SSL/TLS session key S_c is generated (which is never disclosed to WalletHelper) and sealed[11] (together with pre-master secret PMS and the hash value $hash(Cert_S)$). The resulting session-token ST_c is stored in a secure storage and can be loaded and unsealed later using the session identifier SID.

$ChangeCipher$ messages exchanged by WalletHelper and WebServer indicate they are ready to switch to the newly-negotiated parameters and the secret key. All following messages should be encrypted using S_c.

All messages which are exchanged between WalletHelper and WebServer are standard messages of SSL/TLS handshake and can be found in the corresponding specification [16].

Managing two logical SSL/TLS subchannels. To split a single SSL/TLS channel into two logical parts, we utilize the SSL/TLS resume protocol, which allows the re-negotiation of an SSL/TLS session key from the previously negotiated SSL/TLS parameters. For switching the channel to WalletHelper, the SSL/TLS channel is resumed and WalletCore discloses the renegotiated SSL/TLS

[11] Sealing means protecting an object so that only a certain set of entities can access or use it.

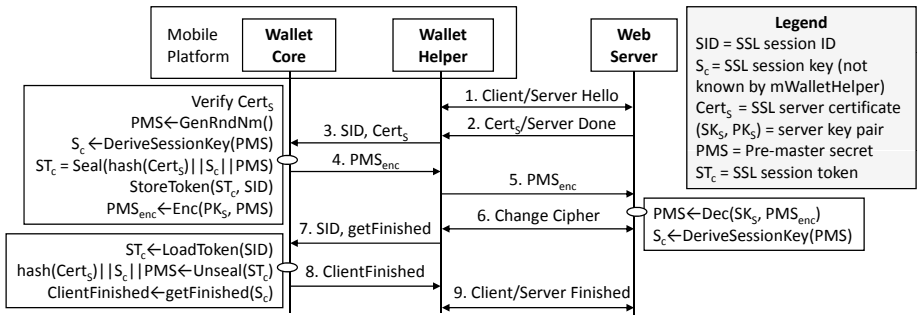

Fig. 2. SSL/TLS handshake

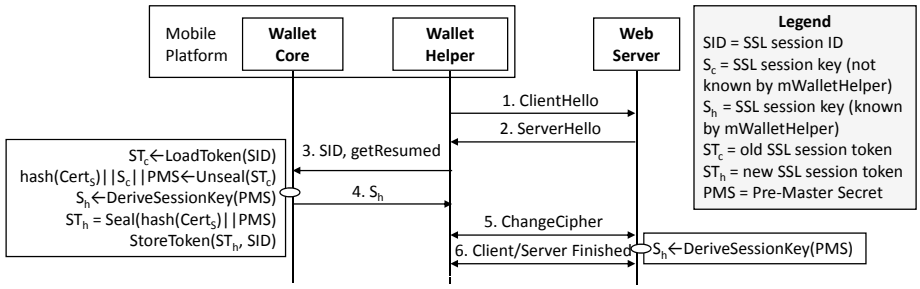

Fig. 3. Resume SSL/TLS connection and switch logical channel to WalletHelper

session key to WalletHelper. Disclosing a new session key preserves backward secrecy, as untrusted WalletHelper cannot decrypt data transmitted previously by WalletCore. The logical channel between WalletHelper and WebServer is switched back to WalletCore by resuming the current SSL/TLS connection and deriving a new session key unknown by WalletHelper.

We depict the protocol of switching logical channel to WalletHelper in Figure 3, and illustrate the complementary protocol for switching the logical channel back to WalletCore in Figure 4. In both diagrams, all protocol messages between WalletHelper and WebServer follow the flow of the SSL resume protocol, as specified in [16]. In the secure side, in both cases WalletCore obtains pre-master secret PMS from an SSL session token and derives a new session key. Note, that the session key S_c is included in the token ST_c, as it is used by WalletCore. In contrast, the session key S_h is not included to session token ST_h, because it is disclosed to WalletHelper and should never be used by WalletCore.

Login/registration/password page forms can be delivered either via http (e.g., Facebook), or via already established SSL/TLS connection (e.g., Google). In the former case, SSL/TLS connection is established when the user presses, e.g., the

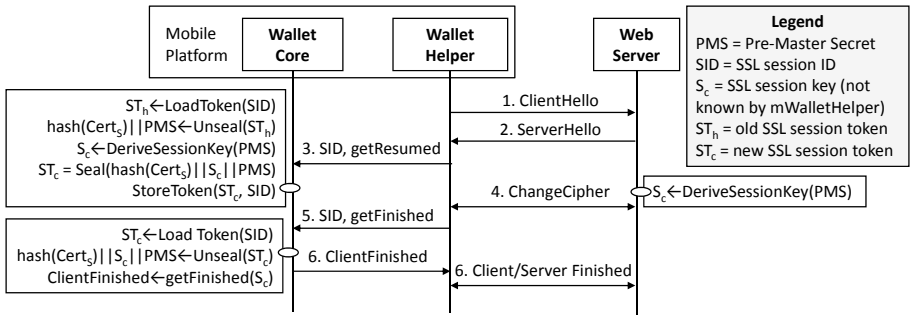

Fig. 4. Resume SSL/TLS connection and switch logical channel to WalletCore

login or register button. To handle this case, $TruWalletM$ establishes the SSL/TLS connection and uses the first negotiated session key for authentication, i.e., the logical SSL/TLS channel is directly established between the WalletCore and WebServer. In the latter case, $TruWalletM$ discloses the session key immediately after SSL/TLS session is established such that the user can surf web-pages delivered via SSL/TLS connection, i.e., the logical channel is established between the WalletHelper and WebServer. However, when a user's credentials have to be sent to WebServer, e.g., after the login button is pressed, the current SSL/TLS session is resumed in order to switch to the logical SSL/TLS channel between WalletCore and WebServer for a secure transmission of the credentials.

4.2 TruWalletM Initialization

When $TruWalletM$ is installed on a mobile platform, a few initialization steps are required: (1) the web browser is configured to work with $TruWalletM$ as SSL/TLS proxy, (2) the user should customize the trusted user interface by adding user-specific interface elements such as a background picture, and (3) the user may want to install passwords for web servers he has already signed up with.

Passwords for existing web accounts are added to $TruWalletM$ by visiting the login page of a web server and pressing the login button. This triggers $TruWalletM$ to search for a password in its database and subsequently to prompt the user with a dialog requesting to specify the missing login and password. We require the user to first visit the login page, instead of directly entering login/password into $TruWalletM$, in order to bind the user password to the server certificate. However, we want to avoid a man-in-the-middle attack during the initialization, e.g., a malicious program that replaces the server certificate by a valid certificate of a different site. Therefore, we follow the approach of [17] and display a list of few destinations (among them the user-requested one) and ask the user to explicitly choose the destination again. As shown in [17] this prevents the user from associating the right credentials to the wrong site.

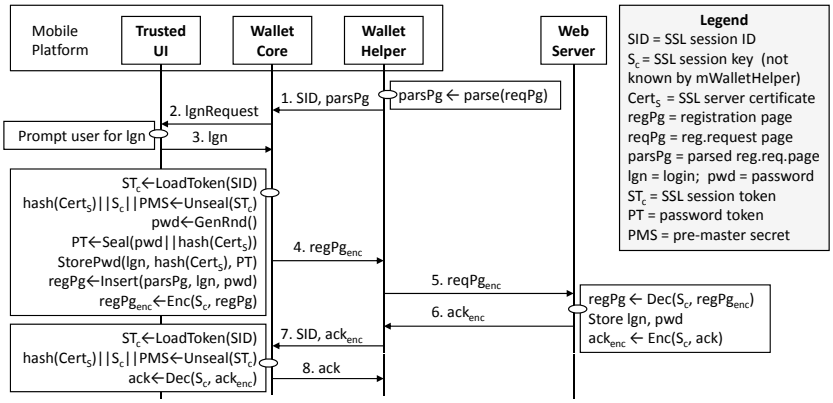

Fig. 5. Registration of a new account

Alternatively, existing accounts can be installed via out-of-band (OOB) channel, e.g., by means of secure provisioning [7]. This protocol can securely deploy credentials, so that they are only known to the provisioning entity and programs executing in the TrEE of the target device. This protocol makes use of a device-specific key-pair (typically available on mobile platforms with TrEE) whose private part resides only in and never leaves the TrEE. Also, OOB channel can be used not only at initialization phase, but later on as well at any time the user wishes to install new passwords.

4.3 Registration

To sign up for a new account, the user visits a registration page of the corresponding web server and presses the "sign up" button. This action launches the registration protocol (Figure 5).

Before the protocol starts we assume a registration request page $reqPg$ is delivered to $TruWalletM$ either through SSL connection or http. At protocol start, WalletHelper parses $reqPg$ and sends a parsed registration request page $parsPg$ to WalletCore (step 1). WalletCore invokes TrustedUI to obtain user login lgn from the user (steps 2-3), loads the SSL/TLS session token ST_c from the secure storage and unseals it. Next, it generates a new high entropy password pwd, seals it together with a hash of SSL/TLS certificate $Cert_S$ (extracted from ST_c) and stores the resulting password token PT in a secure storage. After that, it fills in pwd into $parsPg$ and sends it over the established SSL/TLS channel to WebServer (steps 4-5). WebServer validates lgn and pwd and responds with acknowledgment (steps 6-7). Finally, WalletCore passes the decrypted acknowledgment to WalletHelper (step 8).

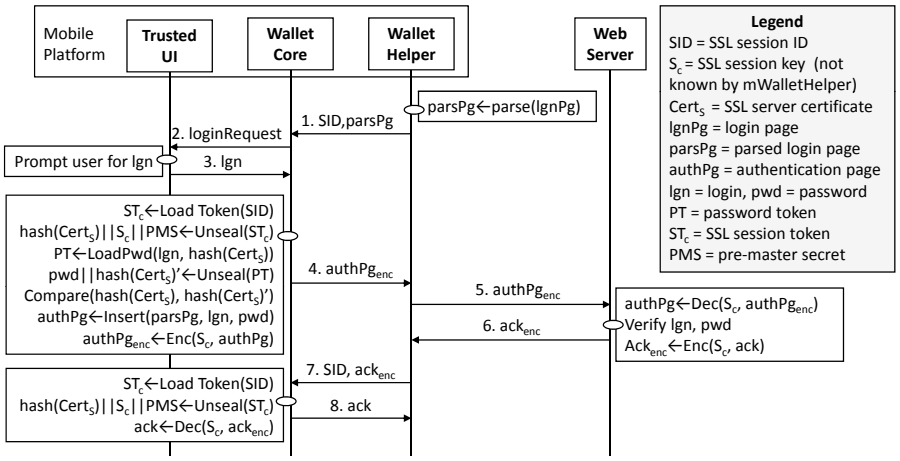

Fig. 6. Password authentication to the web server

4.4 Login

The most typical usage scenario for $TruWalletM$ is to login to a service provided by a web server. To trigger the login procedure, the user visits a login page. The login page is delivered to WalletHelper either via http connection or via SSL/TLS channel established between WalletHelper and WebServer. Next, the protocol proceeds as depicted in Figure 6.

WalletHelper parses a login page $lgnPg$ and sends it parsed, i.e., in a specially prepared format, to WalletCore together with the identifier of the SSL/TLS session, SID (step 1). WalletCore invokes TrustedUI to obtain a login name lgn from the user (steps 2-3). Next, WalletCore loads and unseals session token ST_c, loads password token associated with lgn and the particular web server. Also, WalletCore compares the hashes of the certificates obtained from ST_c and PT to verify the binding of the password to WebServer. If positive, lgn and pwd are inserted into the parsed login page $parsPg$, and the resulting $authPg$ is sent to WebServer encrypted under the SSL/TLS session key ((steps 4-5). WebServer verifies whether the submitted lgn and pwd belong to an authorized user, and replies with acknowledgment on successful verification (steps 6-7). Finally, WalletCore relays the decrypted acknowledgment to WalletHelper (step 8).

Note, if the user is not yet registered at this site, $TruWalletM$ will proceed as explained in Section 4.2.

4.5 Password Change

The last usage scenario we consider is a password change. To change the password, the user visits the password change web-page of the corresponding web server. The page is delivered to $TruWalletM$ either via http connection or via the SSL/TLS channel established between WalletHelper and WebServer.

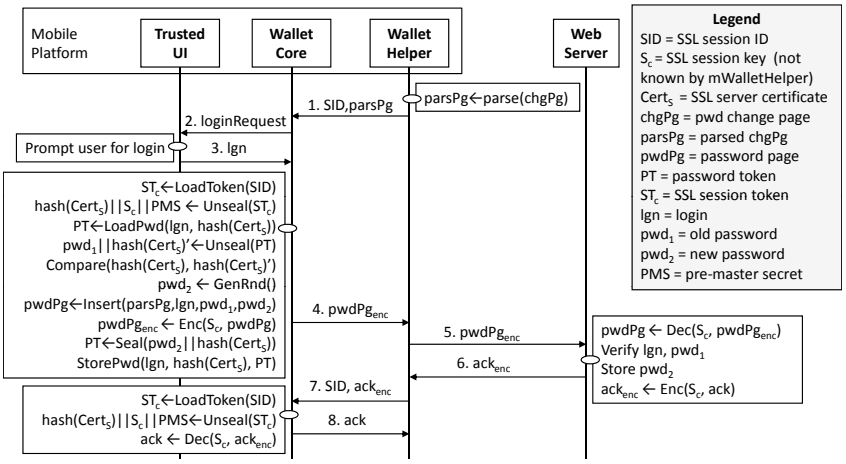

Fig. 7. Password change

Password change protocol is shown in Figure 7. WalletCore gets the parsed password change page $parsPg$ and the identifier of the SSL/TLS session, SID, from WalletHelper and a user login lgn from TrustedUI (steps 1-3). Next, it loads and unseals an SSL session token ST_c and a password token PT. When a password pwd_1 is extracted from PT, its binding to WebServer is verified by comparing the hashes of the certificates obtained from PT and ST_c. If positive, WalletCore generates a new password pwd_2 and fills in lgn, pwd_1 and pwd_2 into the parsed password change page $parsPg$. Next, this page is sent to WebServer via the established SSL/TLS channel (steps 4-5). WebServer verifies whether pwd_1 corresponds to the authorized user, replaces the old password with a new one, and acknowledges upon success (steps 6-7). Finally, the decrypted acknowledgment is passed to WalletHelper (step 8).

5 Implementation

The implementation of $TruWalletM$ is a work in progress. Currently, it is implemented as a user space component, but it is being ported to run in the TrEE of a Nokia N900 smartphone, running Maemo OS [18]. The Nokia N900 emulates the M-Shield secure hardware.

By default, M-Shield is not available to third party developers. However, the On-board Credentials platform (ObC) [7] provides the means to leverage the TrEE. Thus, in our implementation, we build the WalletCore on top of ObC.

A detailed description of the ObC architecture can be found in [7]. In a nutshell, it provides a bytecode interpreter residing in the TrEE, which can execute lightweight bytecode, compiled from scripts written in assembly code. Moreover, the interpreter provides an interface for commonly used cryptographic primitives. Furthermore, the provisioning subsystem of ObC enables in our implementation

the out-of-band provisioning of users' passwords. The sealing functionality provided by the interpreter is used to implement SecureStorage. When sealed, data is encrypted under a key that is never available outside the TrEE. Further, this key is cryptographically bound to scripts, thus isolating the SecureStorage from other, unauthorized (and potentially malicious) scripts.

WalletHelper is implemented as two subcomponents. The first is written in the C programming language and implements the communication with the WalletCore, the interaction with the user, and the parsing of the received login pages. The second component implements the SSL/TLS proxy functionality and is based on the open-source SSL/TLS Java proxy Paros [19].

The WalletCore functionality is currently implemented as part of the first subcomponent of WalletHelper, but will be ported to a set of ObC assembly scripts, which will be invoked by WalletHelper when needed.

The TrustedUI component is currently implemented as part of the OS, because the constraint resources of contemporary TrEEs prohibit a full user interface implementation in the secure execution environment.

We have tested our current $TruWalletM$ prototype with several public websites, such as web e-mail services, eBay, or Amazon. Registration, login, and password change work transparently and without noticeable performance overhead for the user, despite the running SSL/TLS proxy. The performance tests have been implemented based on the open-source *wget*[12] tool, which was used to login via SSL/TLS to the websites *mail.rub.de* and *checkyourbets.com* utilizing $TruWalletM$. The test was performed 10 times. The induced performance penalty for the first website averaged 0.4s and increased the required login time from 1.3s to 1.7s. The performance penalty for the second website was 0.2s and increased the login time from 1.5s to 1.7s. The memory consumption during these tests is acceptable with approximately 60 kB resident RAM, including Paros.

Regarding the performance of our architecture, we do not expect a significant overhead when WalletCore is implemented on top of ObC instead of as a user space process. The performance of bytecode executing on the ObC interpreter is slightly slower compared to native code. This is due to the performance penalty of bytecode interpretation and additional switches from the insecure side to the ObC interpreter on TrEE side. According to our experiments with other similar ObC scripts we can estimate that execution of WalletCore would take 10-20 ms. This cost is negligible compared to the execution time required for the necessary operations during an TLS/SSL connection, e.g., cryptographic operations. Moreover, the invocations of WalletCore will be limited to the steps required to ensure the secrecy of the user's credentials and to the initial steps of the TLS/SSL connection. By providing the session keys for the remaining parts of the TLS/SSL connection to the WalletHelper, the bulk of the operations during the connection will be performed in the insecure environment and do not add any further overhead. Thus, the performance overhead imposed by migrating the WalletCore on top of ObC will be minimal.

[12] http://www.gnu.org/software/wget/

Based on our experience in implementation of other ObC scripts, we envision following challenges in porting WalletCore as ObC script: First, implementing fully flexible X.509 certificate parsing and verification under the constraints of limited TrEE resources has turned out to be challenging. Second, in many cases verification of a single certificate is not enough, but rather full certificate chain verification is needed. Since existing TrEEs on mobile devices are constrained in resources, full certificate chains cannot be verified in one go, but rather chaining of subsequent ObC script executions is needed. The ObC Interpreter provides support for such statefulness. Third, flexible certificate verification implies that multiple trust roots (e.g. hashes of CA public keys) are fixed to the WalletCore script implementation. This increases the size and complexity of the WalletCore script implementation.

6 Security Considerations

Protection of user credentials. The user credentials are protected by the following means: (i) *run-time isolation*. All operations on user credentials (except user input) are performed within the TrEE. As the TrEE is isolated from the rest of the system in terms of processing and memory, that guarantees protection of the user's credentials at run-time from potentially malicious OS and other OS-side components. (ii) *secure storage*. Sealing with a key that is protected by the TrEE and bound to authorized scripts provides secure storage of persistently stored credentials outside the TrEE. When sealed, credentials cannot be unsealed by malware or an adversary performing physical attacks; (iii) *strong passwords*. $TruWalletM$ generates high-entropy passwords which are unique for each account to prevent dictionary, brute-force and reuse credential attacks; (iv) *blind passwords*. Classical and malware phishing attacks are prevented because the passwords are unknown to the user. $TruWalletM$ either creates the passwords within the TrEE and does not reveal them to the user (as proposed in [20]), or, when entered by the user, requests the user to initiate a password change (i.e., to visit a password change page of the associated web server); (v) *tamper-resistant TrEE*. As discussed in Section 2 (assumptions), we assume that TrEE is tamper-resistant against standard physical attacks aiming to access the security sensitive information within TrEE.

Trusted path. The trusted path between the user and the web server is composed of two parts: (i) between the TrEE and the web server, and (ii) between the user and the TrEE.

The first part is provided by the SSL/TLS channel: Passwords are transmitted to the server through an SSL/TLS protected connection, this prevents the adversary from eavesdropping on the communication channel with the web-server. The web-server and the TrEE are mutually authenticated, that prevents man-in-the-middle and phishing attacks. The TrEE authenticates the web-server by validating the SSL/TLS certificate, and the server authenticates the TrEE by means of the password authentication protocol. This part of the trusted path is

secure, because the server certificate is verified within TrEE and credentials associated with the SSL/TLS channel (such as a pre-master secret key and a session key) are protected by the TrEE. Also, passwords used during the authentication protocol are available only within the TrEE.

The second part of trusted path is provided by means of the trusted user interface TrustedUI. TrEE authenticates the user by means of user login upon the start of $TruWalletM$, while the user authenticates TrEE by recognizing a customized interface element, e.g., a unique phrase or background image. This customized element is securely stored in the secure storage and is accessible only by the TrustedUI component. We rely on a customizable user interface since some studies show positive results of such an approach [14], however, this is an issue that requires extensive usability tests since other studies say users tend to ignore security hints [21].

To protect user credentials from malware such as keyloggers, malicious web-browser and phishing programs, the user enters his passwords only via the trusted user interface. However, as in current implementation TrustedUI is realized as OS-side component, we can only guarantee its integrity at system (re)boot (by means of secure boot). Thus, the user can enter his passwords securely via TrustedUI only immediately after system reboot, but later on security guaranties do not hold anymore as TrustedUI is susceptible to runtime compromise.

Discussion on runtime compromise of TrustedUI. In general, a compromised TrustedUI does not provide secure user input from the user to the TrEE, e.g., during the initialization of $TruWalletM$ the entered password can be disclosed by a compromised TrustedUI. Another issue relates to the registration protocol. As it is initiated by an untrusted component, WalletHelper, it can omit the invocation of WalletCore or substitute the server certificate with another valid one for a malicious website, while the compromised TrustedUI indicates a successful and secure execution of the protocol.

The out-of-band provisioning of credentials (as was discussed in Section 4.2) can mitigate these issues in certain cases like the initialization. However, to provide a fully trusted path, either runtime integrity monitoring mechanisms or a TrEE-based user interface are required, which are both currently infeasible for mobile devices. The former either requires extra hardware support (e.g., [22]) or utilizes virtualization technology (e.g., [23]) which is hardly affordable for mobile devices due to the induced overhead and also contradicts to our requirement on software reuse. The latter is infeasible due to the very limited resources of contemporary TrEE.

7 Related Work

As mentioned earlier, on desktop PCs there exist wallet-like authentication agents [1,2,3,4,5] that automatically manage mutual authentication between the user and a web server. To prevent malware from disclosing the credentials based on run-time attacks against the wallet, the wallet is generally executed in an envi-

ronment that isolates it from the rest of the software stack that is responsible for web browsing. This is similar to the TrEE we use in our design here, however, on the PC-based wallets, the isolated execution is achieved by running the trusted and untrusted parts in different virtual machines that are controlled by a trusted virtual machine monitor. For instance, TruWallet [1] uses a virtualization-based security kernel for isolation and a Trusted Platform Module (TPM) [24] to bind the credential data to the wallet. In contrast, Delegate [4] is a web proxy running on a physically different machine and not on the same device the user runs the browser. Besides the protection of the login credentials, SpyBlock [3,11] also protects against malicious software that misuses authenticated sessions to issue illegitimate transactions (so called transaction generators). The trusted authentication agent is therefore extended with a transaction confirmation component. But this design can be applied to all wallet-like solutions, and, thus, to our $TruWalletM$ as well.

To overcome with some of the problems of username/password authentication, alternative authentication protocols have been proposed. For instance, the PAKE protocol for password-assisted key exchange may be used, e.g., [25,26,27]. PAKE protocols can be used for mutual authentication, i.e., it is not necessary to transmit username and password over the SSL/TLS channel during login. The simple remote password (SRP) protocol [27] can be used as well. RFC5054 [28] specifies the use of SRP for SSL/TLS authentication. However, these protocols have the drawback that (i) their security-sensitive operations are not appropriately isolated when used on standard operating systems, and (ii) they are not compatible to legacy web server as they require to change the server logic.

Another alternative is to use challenge-response protocols to verify knowledge of a shared secret between client and server, e.g., as used in TruWallet [1]. However, this again requires to change the authentication verification method at the server, which was one aspect we wanted to avoid in our design.

8 Conclusion and Future Work

In this paper, we present a secure wallet-based system and protocols for protecting user credentials on mobile devices used to access Internet services. Our solution does not rely on a secure operating system, but exploits hardware security features, in particular Trusted Execution Environment (TrEE) available on many modern mobile phones. It is fully compatible with legacy software (e.g., standard browsers and standard OSes) and standard web authentication methods (here password-based authentication and SSL). It does not impose noticeable performance overhead during transfer of conventional data, but slightly slows down communication when transferring user passwords (e.g., during login/registration/password change). We propose a prototype implementation on a Nokia N900 device with M-Shield technology.

Our design assumes the availability of a trusted user interface. Currently this interface is implemented as part of the OS, because the constraint resources of contemporary TrEEs prohibit a full user interface implementation in the se-

cure execution environment. The security issues imposed by this design decision, especially a run-time compromise of the interface leading to credential misuse or potential disclosure, have been pointed out. However, recognizing the trend of device manufacturers to further extend their devices' TrEE capabilities, we argue that our proposed architecture is fundamentally a step into the right direction towards hardware secured user credentials on mobile end-user devices. The challenge of implementing a TrEE-based GUI (and generic I/O inside the TrEE) is orthogonal to the work presented in this paper.

Our further ongoing work concerns the design and development of a confirmation channel and migration protocols. The former assures a user that a certain transaction has been correctly performed while the latter allows the user to securely transfer his credentials from a mobile platform to another platform. This protocol is required to enable users to login from different platforms, because in our solution users do not know their actual (high-entropy) passwords (this is to avoid phishing attacks).

References

1. Gajek, S., Löhr, H., Sadeghi, A.R., Winandy, M.: TruWallet: trustworthy and migratable wallet-based web authentication. In: STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing, ACM (2009) 19–28
2. Gajek, S., Sadeghi, A.R., Stüble, C., Winandy, M.: Compartmented security for browsers – or how to thwart a phisher with trusted computing. In: 2nd International Conference on Availability, Reliability and Security (ARES'07), IEEE Computer Society (2007) 120–127
3. Jackson, C., Boneh, D., Mitchell, J.: Spyware resistant web authentication using virtual machines. http://crypto.stanford.edu/spyblock/ (2006)
4. Jammalamadaka, R.C., van der Horst, T.W., Mehrotra, S., Seamons, K.E., Venkasubramanian, N.: Delegate: A proxy based architecture for secure website access from an untrusted machine. In: 22nd Annual Computer Security Applications Conference (ACSAC'06), IEEE Computer Society (2006) 57–66
5. Kwan, P.C.S., Durfee, G.: Practical uses of virtual machines for protection of sensitive user data. In: Information Security Practice and Experience Conference (ISPEC'07), Springer (2007) 145–161
6. Selhorst, M., Stüble, C., Feldmann, F., Gnaida, U.: Towards a trusted mobile desktop. In: Trust and Trustworthy Computing (TRUST 2010). Volume 6101 of LNCS., Springer (2010) 78–94
7. Kostiainen, K., Ekberg, J.E., Asokan, N., Rantala, A.: On-board credentials with open provisioning. In: Proc. of the 4th ACM Symposium on Information, Computer, and Communications Security (ASIACCS'09), ACM (2009) 104–115
8. Azema, J., Fayad, G.: M-Shield mobile security technology: making wireless secure. Texas Instruments White Paper (2008) http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.
9. Alves, T., Felton, D.: TrustZone: Integrated hardware and software security. Information Quaterly **3** (2004)
10. Security, H.: Hacker extracts crypto key from TPM chip (2010) http://www.h-online.com/security/news/item/Hacker-extracts-crypto-key-from-TPM-chip-927077.html.

11. Jackson, C., Boneh, D., Mitchell, J.: Transaction generators: Root kits for web. In: 2nd USENIX Workshop on Hot Topics in Security (HotSec'07), USENIX Association (2007) 1–4
12. Ristic, I.: Internet SSL server survey. In: BlackHat USA 2010. (2010)
13. Dhamija, R., Tygar, J.D.: The battle against phishing: Dynamic security skins. In: SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security, New York, NY, USA, ACM (2005) 77–88
14. Bank of America: Identity Theft Fraud Protection from Bank of America. http://www.bankofamerica.com/privacy/sitekey (2010)
15. Itoi, N., Arbaugh, W.A., Pollack, S.J., Reeves, D.M.: Personal secure booting. In: ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy. (2001) 130–144
16. Network Working Group: The transport layer security (TLS) protocol. version 1.2. Standards track (2008) http://tools.ietf.org/html/rfc5246.
17. Wu, M., Miller, R.C., Little, G.: Web Wallet: Preventing Phishing Attacks by Revealing User Intentions. In: 2nd Symposium on Usable Privacy and Security (SOUPS'06), ACM (2006) 102–113
18. Maemo: Project website. http://maemo.org (2010)
19. Paros: Project website. http://www.parosproxy.org (2010)
20. Gajek, S., Sadeghi, A.R., Stuble, C., Winandy, M.: Compartmented security for browsers - or how to thwart a phisher with trusted computing. In: ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security, Washington, DC, USA, IEEE Computer Society (2007) 120–127
21. Schechter, S.E., Dhamija, R., Ozment, A., Fischer, I.: The emperor's new security indicators. In: SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2007) 51–65
22. Nick L. Petroni, J., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: Proceedings of the 13th USENIX Security Symposium, USENIX (2004) 179–194
23. Baiardi, F., Cilea, D., Sgandurra, D., Ceccarelli, F.: Measuring semantic integrity for remote attestation. In: Trusted Computing, Second International Conference, Trust 2009, Oxford, UK, April 6-8, 2009, Proceedings, Springer (2009) 81–100
24. Trusted Computing Group: TPM Main Specification, Version 1.2 rev. 103. (2007)
25. Bellovin, S.M., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: IEEE Symposium on Security and Privacy (S&P'92). (1992) 72–84
26. Jablon, D.P.: Strong password-only authenticated key exchange. Computer Communication Review **26** (1996) 5–26
27. Wu, T.: The secure remote password protocol. In: Network and Distributed System Security Symposium (NDSS'98), The Internet Society (1998) 97–111
28. Taylor, D., Wu, T., Mavrogiannopoulos, N., Perrin, T.: RFC5054: Using the secure remote password (SRP) protocol for TLS authentication (2007) http://www.ietf.org/rfc/rfc5054.