Identity in Mobile Security

Layer 7 Technologies

White Paper

Contents

Introduction

In recent years, the concept of "Anywhere, Anytime Computing" has become the common denominator in driving personal electronic device sales, as users are adopting to new categories of devices such as smartphones, tablets, and smartTVs. These devices enable consumers and employees to access information and services from almost any device at any time. Gartner studies show that the estimated mobile phone market is to reach 1.8 billion devices in 2013 [1].

As the surge in mobile device sales continues, the number of iOS and Android applications that provide a more personalized app experience for both professional and private usage is proliferating. To meet high demand, developers worldwide are leveraging social network identity platforms or enterprise identity systems to provide a customized app experience. On top of this, applications and data are now dispersed in multiple datacenters around the globe, and the primary challenge is how to manage the increasing number of user identities that need to securely access these applications [2]. However, protecting one's identity information may not be taken into consideration as often as users would like. In many cases, users need to access different resources that reside in a cloud environment or behind an enterprise firewall. Thus, fortressing identity has superseded the traditional enterprise network perimeter as the new model for security.

The concept of using identity as a basis of access control is not a new digital invention; national passports for example have been an unequivocal source of identity verification for over 600 years. But the ease of transferring information in a

digital and mobile-enabled world has made confidential data management more imperative, especially in mobile applications.

Payment by mobile device is an example of a new and innovative mobile service that relies heavily on verified but sensitive user information. Exactly how any mobile app accepts user credentials and verifies information is a critical success factor. Therefore, two parts exist for this issue: authentication and protection of data. Mobile apps need to resolve and verify user identities in a reliable and trustworthy manner.

The OAuth protocol was introduced to defeat the anti-password pattern where users previously had to share their credentials with apps whenever access to a protected resource was necessary. Even though OAuth has improved the situation, it is still often the case when the user is still required to type out passwords. Consequently, this has lead to an increased usage of low-entropy passwords. A recent study showed that approximately 82% of passwords were cracked within an hour [3]. This is a concern, and as user identity has quickly become the main critical service enabler, which means a stronger focus on mobile security is paramount. Until very recently, the mobile industry as a whole has been mainly concerned with device management, and what has been missing is a renewed focus on enabling secure applications. In collaboration with Samsung, the NSA (National Security Administration) has taken a step toward this direction by creating SE Android. However, this solution is only available through the Samsung Knox program, and does not address a particularly important scenario where the mobile app is consuming sensitive data on the backend.

By looking at the security gaps in mobile applications, the following critical areas must be resolved in protecting user identity and data. First, mutual trust should be established between the client app and the backend API provider. Second, an enterprise or organization's identity management infrastructure must develop a method of assisting mobile apps that require access to resources behind firewalls. Third, the usage of username-password authentication schemes is reduced to a minimum while security rules are still applied.

This paper brings forth recent threats that have affected millions of users and suggests a strong yet simple low-cost solution that not only allows mobile apps to access sensitive data, but retains the trustworthiness of client apps and its users.

Recent Threats

In 2012, millions of Facebook, Twitter and Pinterest user accounts were compromised as attackers launched more sophisticated social engineering attacks than ever before [4]. Recently in early 2013, a large scale attack by a mobile app that disguised as "secure banking software" victimized Android users in several countries, by forwarding personal information to hackers located in Russia [5]. On a smaller scale, some Trojan-horse programs such as Android/MarketpayA made purchases

unbeknownst to the mobile user. These are just some of the threats that have affected mobile users worldwide.

According to the McAfee Threats Reports, mobile malware samples have increased alarmingly, reaching over 50,000 in 2013 with 28% detected just in the first quarter of the year [6]. Simultaneously, as part of a set of recommendations for the federal government, the GAO (U.S. Government Accountability Office) reported that malware has increased by 185% in less than a year [7]. Even though common motivations such as financial gain and acquiring free content have continued to boost the number of security threat cases, malicious spyware and targeted attacks are reported to become more prominent in 2013.

With the increased number of companies allowing BYODs (Bring Your Own Device), employees' personal mobile devices may not contain the same level of security measures that most enterprise administrators would specify in corporate-issued devices. Mobile devices have become more versatile in its use, from providing video calls to paying for coffee at the local café. At the same time, corporate apps have access data that are buried deep inside the enterprise. Many of the so-called BYOD usage programs have made it harder to track device inventory, and there is a shift toward controlling mobile apps and its users. Yet, the most common form for authentication is based on username-password mechanism to authenticate the user. More alarming is that many apps streamline the login process by storing credentials locally in clear text.

More often than not, securing user information is second to device management in the development process; hence, flawed implementations of identity verification protocols are common. Consequently, Man-in-the-Middle attacks where the malicious app can gain access to information from an enterprise or sensitive user data has proliferated.

Challenges in Securing Mobile Apps

There are a plethora of reasons for the increase in security breaches. App developers are generally on a tight schedule to deliver high visibility features, and for that reason, security is usually not the focus of their projects. Simultaneously, the number of new attacks that surface is growing so quickly that it is getting increasingly difficult to properly address them. Ideally, one would expect the various device platforms to provide adequate built-in security. The reality is that although platform security is improving, it is not consistent across all platforms, nor is it sufficient enough to address all security issues.

We identified three common challenges developers face when building secure apps.

1. Protection of APIs

One of the main challenges with mobile apps is to be able to consume the backend APIs while providing adequate access to authenticated and authorized requests. While many solutions point to OAuth as a solution, in reality this is only covering the bare minimum. Different APIs have different requirements; in case of high-value APIs where the protected resource has monetary value or otherwise needs a high level of assurance, one would expect a two-way trust establishment to take place before the API can be successfully invoked. Additionally, regulatory concerns such as HIPAA, PCI Compliance and Sarbanes Oxley, where cryptographically secure levels of trust are required by law, are of prime importance in some very large industry sectors.

A mutual trust needs to be established between the app and the backend. It is only in that situation where the protection of backend APIs occurs and the transaction gets executed properly.

2. PKI (Public Key Infrastructure)

Cryptographic security through digital signatures and encryption is useful in a number of areas. As a fundamental building block in securing applications, PKI systems can establish identities of components in a system as well as in mobile app development that includes devices, apps, and users. More importantly, PKI underpin trust establishment between these entities.

Every entity participating in a PKI ecosystem has a public and private key pair. The public key is embedded in a digital certificate to bind it to the owner of the private key. A Certificate Authority hierarchy is required to manage the issuance and revocation of digital certificates within a PKI system. Both parties in a transaction using digital certificates will trust the Certificate Authority. It becomes extremely important to secure the private signing keys of these Certificate Authorities because if these keys are compromised, the entire trust system will be broken.

Equally, it is a challenge to keep the private key secret in a mobile device that can easily be exposed through malware.

Recently, the number of cases involving exploits, attacks, and tampering of sensitive data calls for a need to improve encryption practices in PKI architectures. Attacks such as the hacking of VeriSign in 2010 and on various CA organizations such as GlobalSign and DigicertMalaysia have raised serious concerns among administrators around the globe [8]. Public key systems are vulnerable to man-in-the-middle attacks. Because these digital certificates carry vital information about identify users, it can be concluded that protecting the authenticity and integrity of the certificate is, indeed, imperative [8].

3. Evolving security standards

The industry and vendors are creating security standards and solutions at an accelerated pace. The evolving standards and solutions can be difficult to understand, implement, and maintain for the average developer. There are numerous technologies available for identification, authentication and authorization. Just in the backend system alone there are an infinite number of standards such as SSL with mutual authentication, FTP Credentials, HTTP Basic, and X.509 Certificates. Additionally there are proprietary single sign-on mechanisms that are relevant in app development at the moment.

The adoption of OAuth has driven new requirements into the OAuth 2.0 standard. Similarly, OpenID Connect is constantly evolving to meet developers' requirements. The rapid developments of these standards leave developers with the options of keeping the support for existing implementations or adopt the new versions. All of these standards and solutions may not necessarily be the app developer's focus when there are other more immediate concerns such as meeting a hard project deadline for getting "something" to market. At that point, there is certainly no time to provide advanced capabilities that can dynamically arm API requests with the appropriate security rigor.

Requirements

Organizations that expose APIs for mobile applications require full control over what threat protection mechanisms are deployed. Yet "full control" must be balanced with a viable model that developers can easily use and understand.

An optimal way to ease the developer effort around security is to offer mobile app developers a client-side SDK that conceals the complexity of security. The SDK must provide a simple and easy-to-use API that enhances the calls to the backend API with the necessary security rigor.

When a new authentication scheme is invented it will be easy to update a client library and backend without forcing developer to understand the details. It provides a clean separation between app logic and security concerns.

The key requirements for this solution include providing mechanisms for mobile SSO (Single Sign-On) and mutual SSL. The purpose of mobile SSO is to reduce amount of the password typing on a device. When an SSO session is used, potentially confusing app behavior is avoided, and the user experience is improved. At the same time, mutual SSL will ensure a two-way trust establishment between the client and the backend API.

On the client side, any key material or tokens must be securely stored. However, this requirement remains a challenge, as independent software vendors continue to rely on the strength of the underlying platform. Furthermore, the architecture must distinguish between the main entities in a mobile security use case user, app and

device. Each may need to obtain an authentication token and a system admin should be able to revoke access to each individual entity.

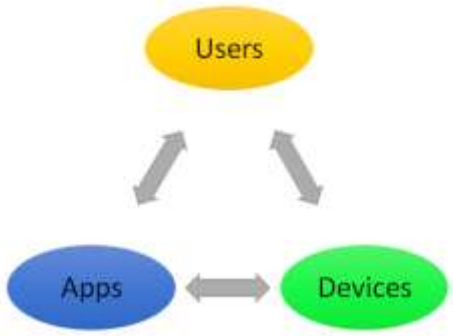

Figure 1: Relationship between the users, apps, and devices

The remainder of this white paper will discuss a solution that uses a combination of OAuth 2.0, OpenID Connect, and PKI to address these requirements.

A new approach to security

In this fresh approach to securing mobile apps, we aim to leverage existing infrastructure as much as possible.

To manage an organization's APIs when exposed to external, partner or internal apps, an API management solution is usually required to easily establish a security perimeter at the edge of your network. The mobile apps tie to a client library that connects the app with the enterprise entry point at the edge. With a simple API call the client establishes a single sign-on session and a secure channel for backend API consumption. See Figure 2 below for an illustration.

Figure 2: An overview of a security perimeter

This approach requires a server side that is capable of basic OpenID Connect and OAuth 2.0, and can support additional protocol extensions.

Server-side Features

To provide a complete solution, a server side component that supports the standard OAuth 2.0 and OpenID Connect endpoints are required. In addition, we propose extensions to these protocols in order to better manage the relationship between users, devices, and apps.

To support a certain degree of security and usability when applications are used with mobile devices, new features must be added to existing protocols. The goal is to provide security and protocol related tasks by the SDK in a way that it is more or less transparent to the developer. Also, it is to give users control of the access of apps and the devices as much as possible without having to call an administrator.

Table 1 below describes this solution in more detail.

Table 1: Mobile SSO - Supported features

Feature	Description
OAuth 2.0	In this proposal for mobile SSO, OAuth 2.0 is required with support for password grant type and JSON Web Token (JWT) Bearer Token grant type.
OpenID Connect (including Mobile SSO related extensions)	OpenID Connect is provided with extensions for Mobile SSO defined as a new scope (msso).
PKI	Signing of CSRs that identify a device and register it for later validations.
Protocol related endpoints	**/connect/device/register**: Registers a device and returns a signed certificate. **/connect/device/remove**: Removes a device registration.
Protocol related OAuth protected endpoints	**/connect/device/list**: Returns a list of devices of a user. **/connect/session/status**: Returns the session status of a user. **/connet/session/logout**: Logout a user and terminates the session. **/auth/oauth/v2/token/revoke**: Revokes a token.
OpenID Connect OAuth protected endpoints	**/openid/connect/v1/userinfo**: Returns claims about a user.

Management-related endpoints	**/msso/manager**: A REST API-based simple view which allows a user to check session status, registered devices, and running apps.

The protocol is using OAuth 2.0, OpenID Connect, JSON WebToken (JWT) and OAuth extensions. This way, the protocol not only enhances but does not break any existing OAuth or OpenID Connect-related implementations. OAuth contains the concept of a shared secret for applications (client_id/ client_secret) for authorization purposes.

Some enterprises do not appreciate this approach since it could be possible for app developers to get access to those values and misuse them. But to address this issue, the protocol allows developers to use mutual SSL which adds strong authentication to the apps. The client-side certificate can be used to authenticate a device or user.

Client Architecture

In order to conceal the complexity of the security and authentication protocols, we propose a client SDK with easy-to-use APIs for mobile app developers. The client SDK will contain a library that the applications, when participating in the mobile SSO session and consume backend APIs, will build against. In order to make a secure sign-in container, the client library will manage the protocol flow and the secure storage of any key material, access tokens, user tokens and certificates. Each app will have its own keychain. The apps that are signed by the same enterprise developer key will use a shared keychain for certificates and user token.

The diagram below illustrates where the applications and certificates are stored, and how they interact with each other on the device.

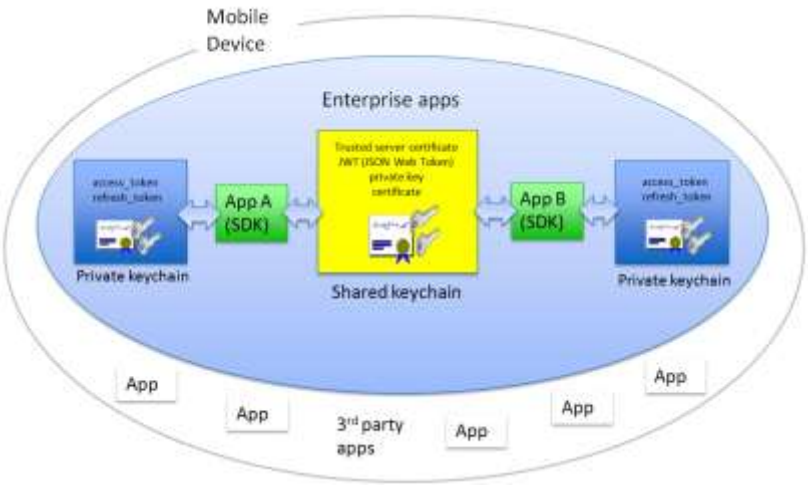

Figure 3: An illustration of the enterprise application architecture

In the mobile device, the enterprise applications area contains only the applications that have been signed by the enterprise. In Figure 3, the applications are marked

with "A" and "B". Each of these applications contains the SDK (Software Development Kit), which handles the OAuth handshake, creates key pairs, and generates a CSR (Certificate Signing Request). The Gateway will only accept the CSR if it contains a unique DN (Distinguished Name).

Each application has access to only its respective private key chain. The private key chain contains the access_token and refresh_token. All the applications within the enterprise area have access to the same shared key chain through the SDK.

In the shared key chain area, the trusted server certificate is imported from the Gateway, which is required when a non-cartel CA (Certificate Authority) is used. The private key is generated and the certificate is signed by the Gateway. The JWT (JSON Web Token) is available in the shared key chain as long as the user is logged into the SSO (single sign-on). Once the user logs out, the JWT is removed.

Client Token and Private Key Storage

The main challenge is to store tokens and keys in a secure manner on the client side. On the iOS platform, bearer tokens and private key material (for example, private exponents for an RSA key) are stored in the iOS keychain when not in use by an app. Apps signed by the same developer key can share secrets among themselves using the Keychain. The contents of the Keychain are encrypted with a key that is entangled with both the device's lock pass code (if any) – this is the 4-digit pin on iOS and the device's secret hardware unique ID.

When a device is locked with a pass code, the key is erased from memory and even if the device's flash memory is cloned, the keychain contents are inaccessible. A simple pass code can be guessed eventually through trial and error but this requires physical possession of the device because the UID cannot be extracted from it. Because this must be done on the device and the hardware is only capable of trying a limited number of codes per second (well under 100), this can take a long time with a complex pass code. Further, this requires plugging the phone into USB hardware, since the iOS software layer can be configured to instantly zeroize the device's memory decryption key after 10 failed pass code guesses. Fishnet Security estimates that a 7-character alphanumeric pass code would take millennia to brute force on-device [9].

On the Android platform, bearer tokens and private key material are stored using the Android Credential Storage mechanism when it is not in use by an app. This service is provided by a system key store daemon that manages a directory that contains encrypted files. Apps signed by the same developer key and declaring the same Android user ID can share secrets among themselves using this mechanism. The contents of the key store are encrypted using a master key that is derived from the device's unlock code. An unlock code must be set on the device in order to use this mechanism.

Screen locks of types "None" and "Slide" are disallowed while credential storage is in use. The Settings activity will not permit these screen lock types to be selected unless the user activates the "Clear credentials" function first. The credential storage master key is erased from memory when the device is locked.

Offline brute force attacks on cloned flash memory are possible but are deterred by the use of PBKDF2 (with an iteration count of 8192) as the key-derivation function. With a high-end GPU capable of computing 4 billion PBKDF2 iterations per second, a master key derived from a pass code with entropy equivalent to a 10-character alphanumeric pass code should take millennia to brute force (though it should be noted that the attack can be split across as many such GPUs as the attacker can afford).

Client-side libraries

Client-side libraries contain easy-to-use APIs for adding the app to a Single Sign-on session. Mutual SSL is established and only a single API call is needed to leverage cryptographic security, OAuth, OpenID Connect, and JWT (JSON Web Token).

Client-side libraries enhance the request with the correct security parameters according to the device configuration. Below is an example of a client library that makes a Single-Sign-on enabled API call with the GET method.

```
L7SHTTPClient *httpClient = [[L7SHTTPClient alloc] initWithBaseURL:
                             [NSURL URLWithString:@"https://www.example.com"]];

[httpClient registerHTTPOperationClass:[AFJSONRequestOperation class]];

[httpClient setDefaultHeader:@"Accept" value:@"application/json"];

NSMutableDictionary * parameters = [[NSMutableDictionary alloc] init];

[parameters setObject:@"listProducts" forKey:@"operation"];

[httpClient getPath:@"/path_to_resources"
           parameters:parameters
                success:^(AFHTTPRequestOperation *operation,
                    id responseObject) {

               //code to handle success response
       } failure:^(AFHTTPRequestOperation *operation, NSError *error) {

               //code to handle failure response

 }];
```

Provisioning Protocol Flow

Below are two diagrams that illustrate how a device receives an access_token.

Figure 4 is a situation where the device is registered for the first time and requests an access token. The area shaded in gray is the application which connects the user to the Mobile Access Gateway.

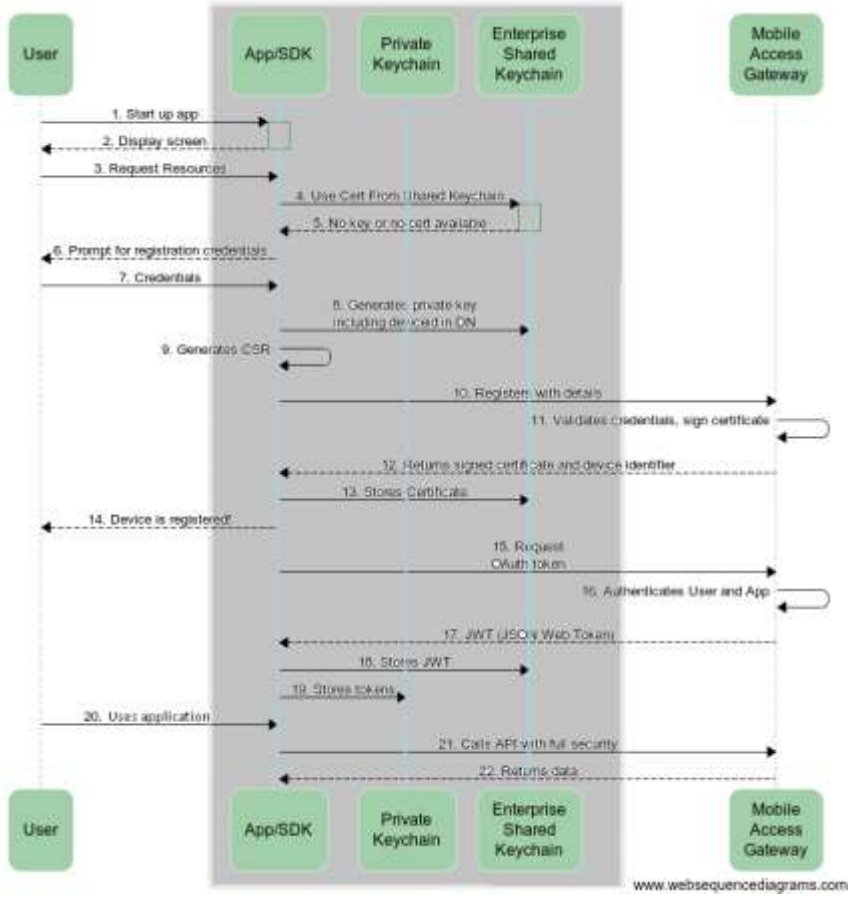

Figure 4: Device registering for the first time, requesting an access token

Since it is the first time the user is registering the device, there are no keys or certificates available. Multiple checkpoints are involved, as mutual authentication and private keys must be established first. The SDK generates a private key and device id to the Enterprise Key Chain. At the same time, a CSR (Certificate-Signing Request) is generated, and then the user profile information is forwarded to the Mobile Access Gateway. Additional proofing may occur at this point. Once the signed certificate is returned and the device is successfully registered, access_token request can be granted.

The second diagram in Figure 5 portrays a simpler scenario where the device has already been registered and mutual authentication has been established. In this case, the registered device requests a subsequent token. The Gateway validates the user with JWT (JSON Web Token) and the application. The access_token is issued once the JWT and application are validated.

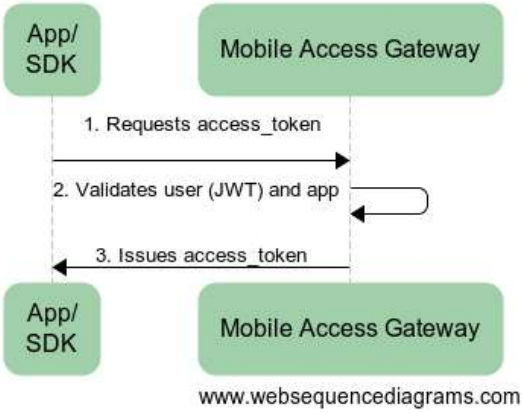

Figure 5: Device already registered and is requesting another access_token

Conclusion

As security threats and identity theft continue to prevail, it is strongly encouraged to place security measures for handling sensitive data as a higher priority in any mobile app development roadmap. The CA Layer 7 MAG 2.0 release, which provides a mobile SSO solution, contains a complete end-to-end standards-based security measure that is easy to implement. With a mobile SDK offering a simple API on the device, any mobile developer can add SSO and secure API calls to backend through the Mobile Access Gateway, without having to be a security expert.

As this area of technology is evolving, we see several promising areas of research.

First is the improvement of aspects within the OAuth flow. Currently, work is being done on the IETF for dynamic registration of clients. This would considerably improve usability, as previously, no interactions existed between parties [10].

Secondly, a more dynamic provisioning of the client configuration could allow enterprises to update a per-app or per-API configuration of requests. As mobile devices change context, the security rigor should reflect the current threat level.

Thirdly, improving the storage of key material and tokens on the device is beneficial. Android 4.2 (also known as “Jelly Bean”) supports key storage using the OpenSSL's ENGINE cryptographic module support, which is capable of using secure hardware key storage wherever hardware and driver support exist. The ENGINE cryptographic module support allows a securely-stored RSA private key to be used for TLS client authorization by utilizing RSA signing, during which the actual private key material never leaves the secure hardware storage or appear in unprotected application RAM (even transiently).

Fourth is investigating how backend applications can trust other sources of identities; for example, from a MDM system or via an external source such as a PIV certificate on a Common Access Card. It would be more cost effective to allow app developers to tie into an organization’s existing identity infrastructure assets. Combining this with a more advanced authentication system can provide real-time, risk-based

authentication that would determine the risk level of online activities. Another benefit is amalgamating user, app and device identification with geolocation and historical patterns, which can be used with custom rules to calculate a risk of each authentication or transaction.

In conclusion, the right strategy for writing mobile apps that leverage backend APIs is to deploy a Gateway function at the edge of the network, and use client-side libraries to insulate the app developer from underlying security protocols. The CA Layer 7 Mobile Access Gateway provides this functionality through open standards.

References

[1] Gartner. "Gartner Says Worldwide PC, Tablet and Mobile Phone Shipments to Grow 5.9 Percent in 2013 as Anytime-Anywhere-Computing Drives Buyer Behavior" http://www.gartner.com/newsroom/id/2525515. June 24, 2013.

[2] CA Technologies. "Identity-centric Security." http://community.ca.com/blogs/iam/archive/2013/05/13/identity-centric-security.aspx. CA Community blog, May 13, 2013.

[3] Dan Goodin,"Anatomy of a hack: How crackers ransack passwords like 'qeadzcwrsfxv1331'."http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/. *Arstechnica*, May 27, 2013.

[4] Sophos. Security Threat Report 2013. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf

[5] "Android Mobile Attacks Spreading across the Globe, McAfee finds." http://www.crn.com/news/security/240155913/android-mobile-attacks-spreading-across-the-globe-mcafee-finds.htm, *CRN magazine*, June 3, 2013.

[6] McAfee. McAfee Threats Report: First Quarter 2013. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2013.pdf?cid=BHP014

[7] United States. GAO. "Information Security: Better Implementation of Controls for Mobile Devices Should Be Encouraged." http://www.gao.gov/assets/650/648519.pdf. September 2012.

[8] "Examining Threats Facing PKI and SSL" http://www.securityweek.com/examining-threats-facing-public-key-infrastructure-pki-and-secure-socket-layer-ssl, *SecurityWeek*, February 11, 2012.

[9] Colin Mortimer. Fishnet Security Blog. "iOS Passwords: Quick Tips to Maximize Your Security." http://www.fishnetsecurity.com/6labs/blog/ios-passwords-quick-tips-maximize-your-security.

[10]. IETP. "OAuth 2.0 Dynamic Client Registration Protocol." http://datatracker.ietf.org/doc/draft-ietf-oauth-dyn-reg/. OAuth Working Group. July 29, 2013.