Emerging Mobile Platforms: Firefox OS and Tizen

Olga Gadyatskaya, Fabio Massacci, Yury Zhauniarovich

Department of Information Engineering and Computer Science,
University of Trento, Italy
e-mail: name.surname@unitn.it

Existing mobile platforms landscape evolves very quickly, as the big players in the field and the research community are challenged to develop novel solutions with minimal costs of application development and possibility to support natively mobile web applications. This process to a great amount has been driven by the presence of Android, an open-source operating system developed by Google. In this article we overview two promising platforms that are both open-source and web-based: Firefox OS (contributed by Mozilla) and Tizen (backed by Samsung and Intel), and compare their security features with those of the Android OS.

I. Introduction

Emerging mobile operating systems target the booming market of mobile web applications (webapps). A traditional mobile webapp is essentially a web page (HTML, CSS and JavaScript) that is accessible through a special web rendering component of a smartphone. Webapps can be easily distributed across multiple platforms, as only a limited fraction of code requires rewriting. Their only (and severe) limiting factor was the lack of access to the native device features, such as the GPS sensor or accelerometer. In order to address this limitation, W3C and a number of companies, including Mozilla, push to standardize a set of client-side APIs that could make the device hardware available to webapps. These client-side APIs provide access to, e.g., the device filesystem, settings, location, telephony services, calendar and alarm mechanism.

With the development of the client-side web API standards, new mobile platforms have emerged. These platforms offer a lightweight mobile application distribution channel via web, while attracting the end-users with relatively cheap devices. This article presents an overview of two novel mobile platforms, Firefox OS and Tizen, compares their security features with those Android OS, and attempts to understand which lessons were learned by the mobile platform manufacturers. We assume the reader is familiar with the Android security architecture (the paper [1] by Enck et al. might be a good starting point to discover Android).

For the scope of this article we use the following namings:

- *Native application* – an application (app) developed in a platform-specific language (e.g., a Java app for Android)
- *Webapp*, or *HTML5 app* – a stand-alone application for a specific mobile platform developed using the web technologies;
- *(Browser-based) web page* – an application/web module that is rendered in a (mobile) browser;
- *Device APIs* – a set of APIs on a specific mobile device that enables webapps to access the device capabilities;

II. Firefox OS

Firefox OS is a Linux-based mobile operating system supported by Mozilla. In the nutshell, Firefox OS is a modification of the Android stack to run Gecko, the application runtime of this operating system (which is also the basis of the Mozilla Firefox web browser).

Currently, Firefox OS is still in its early days, but some hardware companies, including ZTE, LG and Huawei, have already announced the appearance of devices based on this operating system, while GeeksPhone already sells Firefox OS phones. Additionally, Firefox OS was ported to Android smartphones (for instance, to Galaxy Nexus and Nexus S), although they are not considered as primary targets for Firefox OS.

A. The Firefox OS Architecture

Firefox OS relies a lot on the architecture and core principles of its predecessor, Android. Fig. 1 depicts the architecture layers, each with its own name, and the main components of Firefox OS. After looking at the firmware code, one can conclude that in essence the Linux kernel of Firefox OS dubbed Gonk is nothing more than an "Androlyzed" Linux kernel. Gecko is the main app runtime and corresponds to the Android Framework layer. Gaia is the user interface layer entirely implemented using web technologies.

Gecko is a framework that provides the device functionality to webapps in Firefox OS. During the system startup the first user-space process `init` starts the *system core process* called `b2g` (short for "boot to Gecko", the original name of the Firefox OS project), which is also known as the *parent* or *master* process. This process mediates all accesses to the system resources including filesystem access. This means that webapps do not directly interact with the operating system. The communication between the child and the parent processes occurs through an IPC mechanism described using the IPC Protocol Definition Language (IPDL) (that is similar to AIDL in Android). In Firefox OS the APIs available to webapps are defined using this language. Thus, a webapp can interact with the system resources only through the set of well-defined APIs (the Web API). This architecture serves Firefox OS in several aspects. First, it provides an API set, which can be used by child processes to access the features provided by the parent

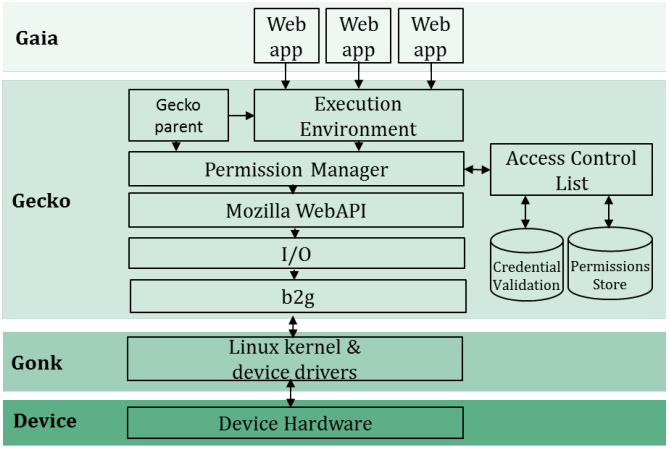

Fig. 1: The Firefox OS architecture [2]

process. Second, it serves as a centralized policy enforcement point, where permissions of the child processes are checked. Additionally, this architecture provides a centralized solution for handling race conditions (when several child processes request the same system feature) [3].

This architecture allows Firefox OS to have security implemented on several layers. On the Gonk layer, it provides application sandboxing using discretionary access control (DAC) implemented in the Linux kernel. On the Gecko layer, it provides mandatory access control (MAC) implemented in the form of permissions.

B. App Ecosystem

For its operation, Firefox OS relies only on HTML5 webapps that are divided into three categories: *web*, *privileged* and *certified*. Certified apps are the apps that are pre-installed on the device (supposedly by phone manufacturers or telco operators). These applications are located on the `system` image. Generally, certified apps correspond to Android's system applications that receive access to the API protected with the `signatureOrSystem` permissions. Similarly to Android, only certified apps may receive access to the "dangerous" functionality, e.g., to the telephony stack or webapp management (non-certified apps cannot detect the presence or any changes to the state of any other particular app).

Privileged apps are webapps that are verified and signed by the app marketplace. Prior to signing an app, the marketplace should perform a code review process, when app authenticity and integrity are verified, the requested permissions are reviewed, and the code is checked for absence of malicious actions. Unlike Android, where the app signature is used to enable trust relationship between applications and to verify that updates come from the same developer, in Firefox OS the marketplace signature is used to elevate the available privileges of the application. Basically, privileged apps are similar to Android third-party apps signed with the system signature (the `signature` permission level) – they also get higher access to the API.

Web apps are similar to ordinary web pages accessible through mobile browser. These apps may have resources stored on the device or located on a web server. According to Mozilla, most third-party apps on Firefox OS will be web apps, because they are the cheapest to develop and maintain. They could be distributed through a website and, thus, do not require an update process. Moreover, the one of the most interesting features of Firefox OS, the ability to work with apps returned by a search engine, works only for web applications. However, web apps have the least set of available permissions.

According to the distribution method, apps can be classified as *hosted* and *packaged*. Hosted apps act like browser-based web pages. All resources of these apps, except the manifest file that is stored locally, are located on an external web server. In contrast, packaged apps contain all their resources (HTML, CSS, JavaScript, manifest, etc.) archived in a zip file. Web apps can be either packaged (in this case they are called *plain packaged* apps) or hosted. At the same time, privileged and certified apps can be only be distributed as packaged apps.

C. Sandboxing

In Firefox OS the apps run in independent sandboxes completely isolated from each other. The sandboxing mechanism is implemented similarly to Android. This technique is based on the well-known DAC model implemented in the Linux kernel. In particular, when the b2g process starts a new web application, the webapp process is run with its own unused, low-privilege user ID (UID). This allows Firefox OS to create a simple powerful sandbox preventing applications from reading each other data. Furthermore, each content process may be additionally sandboxed using the `seccomp` mode [4]. If a process runs in this mode it has access to a limited set of system calls (usually, `sigreturn`, `exit`, `read` and `write`). If the process does other system calls, the Linux kernel simply kills it. Additionally, all interactions of the webapps with the system (including the filesystem access) occur only through the exposed API calls. Interactions of a webapp with a filesystem occur on the whitelist basis, i.e., the *master* process limits the access of the *child* one only based on the list of allowed locations. Moreover, such architecture prohibits direct interactions between applications. Thus, all interactions between applications are only indirect, i.e., they happen through the *parent* process.

This sandboxing solution provides an effective way to separate the data, like cookies and local data (e.g., IndexedDB), stored by different applications. This means that two applications installed on a device will receive different cookies, local data, etc., even if they both open an `<iframe>` pointing to the same origin. Unfortunately, this may create additional storage overhead (for instance, in case when a web page opened in both applications stores a lot of data). However, as a benefit this architecture prevents some attacks from malicious web sites.

D. Permissions

A webapp running on Firefox OS may act as a simple web application opened in a web browser of a desktop operating system. However, to exploit the full potential of a smartphone sometimes it may require access to some device features as,

for instance, contacts database, telephony, or location of the user. At the same time, access to these features may pose additional threat to user's privacy and, thus, should be tightly controlled.

App access to the protected API is restricted through permissions. For each type of permission and application Firefox OS specifies one of the three default actions: `DENY`, `ALLOW`, and `PROMPT`. For instance, for a permission `device-storage:music`, which guards the access to music on the device, the default action for web apps is `DENY`, for privileged is `PROMPT`, and for certified is `ALLOW`. The `DENY` action prohibits access of the app type to the feature. Permissions with the `ALLOW` action are granted on Firefox OS at the app install time. The user cannot grant or deny such permissions; she can only remove the app as a whole (as in Android). The most interesting is the `PROMPT` action. This action requires explicit user's consent at the time of the first feature use; it may be granted or denied permanently. Additionally, at any time the user may revoke this permission through the system settings application.

A Firefox OS permission may have the optional property called `access`. Currently, this property controls the type of access (e.g., `readonly`, `readwrite`, `readcreate`, or `createonly`) to data. This allows developers to specify more fine-grained control over the data.

Another requirement for an app willing to be granted with permissions is that it has to declare in the manifest file the permissions it wishes to receive, along with a textual "justification". The latter in the future will be used to explain to the user why she should grant the requested permission (this feature is still under discussion [5]).

It is worth mentioning that some permissions are implicitly granted to any application (e.g. network access). These are based, according to Mozilla, on what is already available to standard browser-based apps, or are innocent enough to be granted to anyone. At the same time, on Android access to network capabilities is considered as a dangerous permission, because in this case an application may leak user's sensitive data. It would be interesting to re-run on Firefox OS the same experiment by Enck et al. on Android [1] to see if tuples of dangerous permissions could be granted.

Permissions granted by the user are segregated according to their origin: e.g., if an application has received the permission to access geolocation, all scripts of an external origin loaded by this application will still have to request the user for it; this measure protects against privilege escalation attacks.

E. Additional security mechanisms

Gecko also enforces a Content Security Policy (CSP). In essence, CSP is a declarative policy notation to express the expected sources of content loaded in the page [6]; it can be useful for prevention of cross-site scripting attacks. CSP is enforced by the client JavaScript engine, e.g., by restricting loading of resources from outside these sources. Firefox OS apps can express their CSP in the manifest files. Additionally to the declared CSP policy, privileged and certified apps are subjects to additional CSP restrictions enforced by the platform [7].

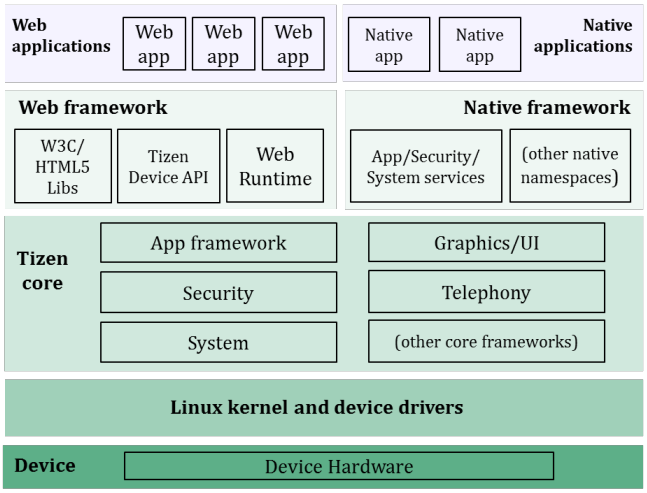

Fig. 2: The Tizen architecture [8]

Another additional security mechanism implemented in Firefox OS is that an app may request two sandboxes: one for itself and another one for any website it accesses. The second sandbox is needed to prevent the accessed content from attacking the resources of the original app. This feature is widely used if a main application is used to visit lots of web resources (e.g., a web browser).

III. Tizen

Tizen is an open-source Linux-based mobile operating system. It is supported by the Linux Foundation and by a number of companies e.g., Intel, Samsung, Huawei, Fujitsu, and telecommunication operators (Orange, Vodafone, etc.).

Initially, Tizen was developed as a web-based operating system, i.e., for webapps only. However, starting from the version 2.0 it is possible to run native applications (developed in C++) on this platform.

A. The Tizen Architecture

Unlike Android, which adopted only the kernel from Linux, Tizen more resembles a Linux distributive. It has the X11 window system with the Enlightened window stack manager. Moreover, it includes a lot of system tools and utilities, such as `ssh` and `sshd` (secure shell and secure shell daemon, respectively), `scp` (secure copy), `bash`, and `rpm` (package manager).

Fig. 2 depicts the Tizen architecture. The lowest layer represents the Linux kernel and device drivers. The higher *Core* layer provides basic functionality required by the Web and the Native frameworks.

The Tizen Web framework provides a possibility to develop and run applications using HTML5 functionalities (referred to as W3C/HTML5 APIs) defined by W3C and other standardization groups. It includes new device APIs (Tizen Web Device APIs), which enable access to the device capabilities, such as Bluetooth, Near Field Communication (NFC), messaging, etc. The runtime environment for webapps (the Web Runtime) is based on WebKit. The Tizen Native framework is composed

by system services and a set of native libraries for development of native applications in C++.

The top layer in Fig. 2 represents user applications. Combining the frameworks used for app development all Tizen apps can be divided into 3 categories: *native* (developed using the Native framework), *web* (designed using the Web framework) and *hybrid* (developed combining both frameworks).

B. App Ecosystem

Tizen in fact supports 3 types of webapps: *mobile website*, *hosted* application and *packaged* application. Mobile website is a simple web application, which is opened in the Tizen web browser. This type of applications has access only to the resources available to an ordinary web page opened in a browser. A hosted app is a client application, which provides access to remote resources. A packaged web application contains all its resources inside its package. The latter two types of apps can be published in the Tizen store. The hosted applications do not have access to the Device API. The standard webapp type on Tizen is a packaged webapp. Only packaged apps can access the Tizen Web Device API.

Native applications are divided into two categories: UI applications and service applications. Both have the same level of access to the API of a Tizen device. However, the main difference between them is that the former has graphical user interface, while the latter does not (it simply runs in the background).

Similarly to Android, a developer may also declare in the manifest the software or hardware features required by the app in order to filter it in the Tizen store. This functionality is missing on Firefox OS.

C. Sandboxing

In Tizen, all the processes are run with one of two UIDs, which correspond to the *root* or *app* account. All application processes run with the app UID. Sandboxing of applications in Tizen is enabled by the SMACK (Simple Mandatory Access Control Kernel) module of the Linux kernel. SMACK is a Linux Security Module (LSM), which enforces MAC on the Linux kernel level. In the model implemented by this LSM, each process (subject) and resource (object) are tagged with special labels. Access of processes to resources is controlled through simple rules represented in the form {*subject, object, permission*}.

To provide effective isolation of applications, each app executable by default is assigned with its own unique 10 characters label (this label is stored in the `SMACK64EXEC` extended file attribute property of the executable). All application resources are assigned with the same label. These labels are assigned to files during the app installation process by the package manager. By default, SMACK rules are defined in a way that the application has full access to its resources. The package manager can also define new rules during the app installation.

Some system objects are also SMACK-labeled. An application may receive access to these protected system objects, if corresponding rules are defined This access corresponds to a *privilege* an application may request to gain access to the additional functionality. During the installation time the package manager may add these rules if the application has passed some security checks.

The SMACK-driven sandbox is applied to all application processes: web, native and hybrid. To provide sandboxing for web applications during the installation of a webapp in its `bin/` folder a soft link to the Web Runtime client is created and labeled with the AppId SMACK label. Thus, when the webapp is started Tizen runs the soft link, which corresponds to the called app, and the web application is executed in a sandbox.

The sandboxing model implemented in Tizen differs from the one implemented in Android and Firefox OS. The latter ones use the Linux DAC to sandbox applications, while Tizen exploits the SMACK LSM, which sandboxes processes using a MAC mechanism. Recently, Android has been also hardened with a mandatory access control module called SEAndorid (based on SELinux) [9].

Native (and hybrid) applications may interact with each other. The Native framework provides 3 main mechanisms for inter-application operations. Apps can export functionality to be available to other applications and query *AppManager* for available functionality of other apps through the *AppControl* mechanism. This mechanism is very similar to *Intents* provided by the Android OS. As on Android, there are two main types of AppControl resolution: *explicit*, when an application calls another app through a unique *AppId* (corresponds to a package name in case of Android), and *implicit*, when AppManager is responsible for providing suitable application, which may process the request based on the operation ID and the URI or MIME type of the data (similarly to the Android's implicit intent resolution). As on Android, by default, each application has the *main* implicit AppControl, which is used to start the application. Another way to communicate is exporting/importing data via the *data control* mechanism (akin to content providers on Android). Apps can also exchange data through *messaging ports*. Bi-directional communication is bootstrapped by exchanging the port references.

A developer can organize private communication among her apps. Similarly to Android, this private communication may be organized through app or data control mechanisms. To perform this, the developer must give permission by application certificate. In this case, an application, which is signed with a different certificate, will have no access to the exported functionality. This mechanism is similar to the Android "signature" permission level protection. Having a lot of in common, the Android mechanism is more fine-grained because it provides possibility to limit the access only to selected components. Additionally, trusted communications in Tizen may be performed through a trusted messaging port or a trusted shared directory. Both approaches require that the communicating apps are granted permissions for trusted communication and share the same signature.

D. Permissions

Similarly to Android and Firefox OS, the sensitive APIs accessed by the app are to be specified in the manifest. On

Tizen the requests to access the sensitive API are defined in the form of *privileges*. Privileges in Tizen are similar to permission declarations in Firefox or Android OS. As mentioned in Section III-C, some system objects are tagged with SMACK labels. Thus, privileges specify the requested access type of an application to the system resource. If an application is granted with the privilege during installation, this privilege is transformed into the corresponding SMACK rule.

Tizen privileges are categorized into three levels. *Public* privileges are available to all apps (e.g., location or contact list access). *Partner* privileges can only be requested by apps coming from companies registered as trusted partners on the Tizen store. Examples of such privileges are access to the device app manager, secure element, system manager, etc. Finally, *platform* privileges (e.g., to access the package manager or settings manager) are available only to the developers working for the Tizen consortium.

The pertinence of an application to the corresponding accessible privilege level is defined by the signatures that are supplied with the app. Unlike to Android and Firefox OS, a Tizen app may be signed with two signatures: the *developer* (or author-signature) and the *distributor* signature (which may be applied by the marketplace vetting applications, or by the device manufacturer, who signs his own system apps). The developer signature determines the authorship; it can also be used to grant additional permissions to applications with the same signature. The distributor signature is used to mark the origin of the app provision (the Tizen store by default). Upon app installation the package manager compares the signatures bundled in the application with the ones stored in the system for different levels. Based on the result of comparison it defines the maximum privilege level available for the installed app.

The access of applications to the device API with different privilege levels is not hardcoded. Instead, it is defined in a Tizen Policy configuration file. Thus, the device manufacturers and telco operators may reconfigure it according to their needs.

E. Additional security features

Similarly to Firefox OS, Tizen allows webapps to express their CSP and restrict the sources of loaded web content. Furthermore, webapps navigation is limited to the list of domains specified in the `<tizen:allow-navigation>` tag of the manifest file.

Additionally, on Tizen a webapp can explicitly turn on encryption, and all resources of the app stored by the device will be encrypted. When this app is launched, the platform will decrypt its resources in a transparent manner.

While on Android there is no standard way to check easily whether a given package contains an invocation of a sensitive device API, the Tizen Native IDE checks for potential problems with missing permissions and API incompatibility.

IV. Lessons Learned and Conclusions

Although Firefox OS and Tizen are still in the beginning of their way, some aspects of their security architectures are very interesting and can be borrowed by more mature systems like Android. Likewise, existing proposals for hardening Android security can be used by Firefox OS and Tizen. For example, starting from the version 4.2, Android fully supports the memory management security enhancements, such as ASLR (Address Space Layout Randomization) and DEP (Data Execution Prevention). At the same time, these protections at the time of writing are not implemented in Firefox OS (implementation of the ASLR feature seems to be in the active phase [10]). As Tizen supports development of apps in native languages, it is more vulnerable to buffer overflow attacks, which can be mitigated using the memory management security enhancements. However, at the time of writing DEP does not work on Tizen, and ASLR is implemented only partially [11].

One of the main differences is the sandboxing approaches used in these systems. As Firefox OS is built on top of Android, these two systems exploit the Linux DAC to sandbox applications assigning each app a separate low-privileged UID. This approach allows to provide app isolation separating the address space of processes. At the same time, the Linux MAC (through SMACK policies) is used to sandbox apps in Tizen, while all apps have the same UID. Properly implemented, the MAC approach is considered as more secure solution. Not surprisingly, adoption of the SELinux MAC has been recently proposed for Android [9], which helped to improve Android security in different aspects (including prevention of root exploits). Yet, SELinux is not used on Firefox OS, and we believe that adoption of this framework can significantly improve its security. For instance, currently `b2g` (the *parent* process) is run with root privileges. Thus, if an adversary finds a way to exploit this process, she will gain full access to the system. At the same time, an exploit execution from the application processes is quite hard on Firefox OS due to the `seccomp` mode usage. This type of hardening can be also implemented in Android and Tizen, accounting the fact that the apps on these platforms may run native code. Considering the Tizen's sandboxing approach, it could be improved in the direction of assigning diffent UIDs to the apps. This will lead to the separation of address spaces of different apps. The combination of DAC and MAC mechanisms in Android could be considered as a reference implementation for Tizen and Firefox OS.

The Firefox OS and Tizen mobile platforms take different approaches to app isolation. While Firefox OS tries to isolate apps completely (providing no direct interactions between apps), Tizen apps have extensive means of interaction (following the Android approach). Both models have proved to be acceptable by end-users, yet the system with interacting apps can potentially expose more vulnerabilities, including the privilege escalation attacks (very relevant for Android [12]); this aspect of Tizen needs to be investigated further.

In the process of permission procurement Android exploits the "all or nothing' approach', i.e., either all permissions are granted during the installation of an app or it will not be installed. However, it has been shown that this approach introduces additional threats to user's private data. We believe the solution used in Firefox OS ("prompt" permissions) is more user-friendly. For Android several similar research solutions have been proposed, e.g. [13], and in the recent

versions of Android a similar feature called *Apps Ops* has appeared. Unfortunately, Google later claimed that this feature was intended to be used only for testing purposes and removed its support in Android 4.4.2 [14].

However, even prompting user for each individual permission may not solve issues with overprivileged apps or apps that receive access to a combination of sensitive functionality. Tools like Kirin [1] may prove useful for Tizen and Firefox OS. One of big concerns for mature mobile platforms is the sensory malware apps that exploit access to device sensors to mine, e.g., password information or credit card numbers [15]. Compared to Android, Tizen and Firefox OS have not raised their guards against this type of malware because they as well do not protect all device sensors with permissions. We believe that more measures of protection are needed for Android, Firefox OS and Tizen to protect users against this type of malware.

Due to the openness of Android ecosystem, at first Google did not distinguish any particular market. However, with the proliferation of mobile malware a strong need for app vetting has appeared. Recently the Google Bouncer tool, which performs application security testing, has appeared. At the same time, the Google's approach does not allow them to distinguish the vetted applications in any way except publishing them in the Google Play store. In contrast, in Firefox OS and Tizen the vetted applications can receive higher privileges comparing to unchecked ones. The approach of Tizen of two signatures used to sign an app is the most interesting: the developer signature verifies the authenticity of the developer and the integrity of the app, and the market signature shows that the application has passed a vetting process and can be considered benign. In Firefox OS, only approved apps receive the market signature, all other apps are not signed. The signature question for Android apps was recently explored in the research community, and a possible solution, which incorporates the benefits of multisigning without breaking current app ecosystem, was proposed [16].

Another problem, where the market vetting process may help, is overprivileged applications. On Android a developer should assign necessary permissions by herself, the Android tools do not help to select necessary permissions. Not surprisingly, a lot of apps are overprivilged [17]. To help developers, the Tizen tools provide a possibility to check, what permissions are required to run the written code. This helps developers to choose only necessary permissions, however, does not solve the problem of overprivileged applications. The same problem is also valid for Firefox OS. To our point of view, marketplace vetting with the tools, which analyse permission usage [17] can help to solve this problem.

Our findings presented in this article can be summarized that emerging mobile platforms indeed take into account some lessons learned from insecurities in Android and other mobile platforms. Yet there are many improvements that can be done.

REFERENCES

[1] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *ACM Conference on Computer and Communications Security (CCS'2009)*. ACM, 2009, pp. 235–245.

[2] Mozilla, "Firefox OS Security Overview," At https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS/Security/Security_model Accessed in December 2013.

[3] ——, "Firefox OS architecture," https://developer.mozilla.org/en-US/Firefox_OS/Platform/Architecture, 2013.

[4] J. Corbet, "Seccomp and sandboxing," http://lwn.net/Articles/332974/, 2009.

[5] Bugzilla, "Bug 823385," https://bugzilla.mozilla.org/show_bug.cgi?id=823385.

[6] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th international conference on World wide web (WWW'2010)*. ACM, 2010, pp. 921–930.

[7] Mozilla, "Apps CSP," https://developer.mozilla.org/en-US/Apps/CSP, 2013.

[8] Tizen, "Tizen Developer Guide," At https://developer.tizen.org/documentation/dev-guide. Accessed in December 2013.

[9] S. Smalley and R. Craig, "Security enhanced (SE) android: Bringing flexible MAC to android," in *20th Annual Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[10] Bugzilla, "Bug 777948," https://bugzilla.mozilla.org/show_bug.cgi?id=777948.

[11] S. Suzuki, "Tizen security," http://www.ffri.jp/assets/files/monthly_research/MR201305_Tizen_Security_ENG.pdf, 2013.

[12] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *19th Annual Network & Distributed System Security Symposium (NDSS'12)*, Feb 2012.

[13] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '11. New York, NY, USA: ACM, 2011, pp. 49–54. [Online]. Available: http://doi.acm.org/10.1145/2184489.2184500

[14] G. Sims, "App ops what you need to know," http://www.androidauthority.com/app-ops-need-know-324850/, December 2013.

[15] R. Schlegel, K. Zhang, X.-Y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *NDSS*, 2011.

[16] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo, "Demo: Enabling trusted stores for Android," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1345–1348.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.