Data-Centric Security for the IoT

Daniel Schreckling[1(✉)], Juan David Parra[1], Charalampos Doukas[2], and Joachim Posegga[1]

[1] IT-Security, University of Passau, Passau, Germany
ds@sec.uni-passau.de

[2] Future Media Area, CREATE-NET, Trento, Italy

Abstract. This work presents a paradigm shift and introduces a data-centric security architecture for the COMPOSE framework; a platform as a service and marketplace for the IoT. We distinguish our approach from classical device-centric approaches and outline architectural as well as infrastructural specifics of our platform. In particular, we describe how fine-granular and data-centric security requirements can be combined with static and dynamic enforcement to regain governance on devices and data without sacrificing the intrinsic openness of IoT platforms. We also highlight the power of our architecture, converting concepts such as data provenance and reputation into efficient, highly useful, and practically applicable complements.

Keywords: Internet of Things · Information flow control · System security · Reputation · Provenance · Identity management · Static analysis · Node-RED

1 Introduction

COMPOSE is an FP7 EU funded project targeting at the development of a full end-to-end solution for developing Internet of Things (IoT) applications and services: from mobile apps for users interaction, to connected objects that sense or interact smartly with the environment, to a scalable data streaming and processing infrastructure, to service discovery, composition and deployment of applications. The logical architecture of the COMPOSE platform is depicted in Fig. 1. Its main components are the COMPOSE Marketplace, the runtime engine, and the Ingestion layer.

The Marketplace is the front-end interface to developers for the publication, exchange, and access of reusable services. It consists of a graphical interface for creating application logic and offers mechanisms for the discovery of existing services, registration of new ones, and the deployment of applications.

Applications are executed on the second layer, the runtime, which is transparent to the developers and provides interfaces for monitoring and support for usage analytics. It is based on an enhanced version of CloudFoundry [3], an already established, open-source PaaS solution with a large community supporting its development. It provides the essential environment for hosting applications, in our case Node.js that we are using for workflow execution.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2016
B. Mandler et al. (Eds.): IoT 360° 2015, Part II, LNICST 170, pp. 77–86, 2016.
DOI: 10.1007/978-3-319-47075-7_10

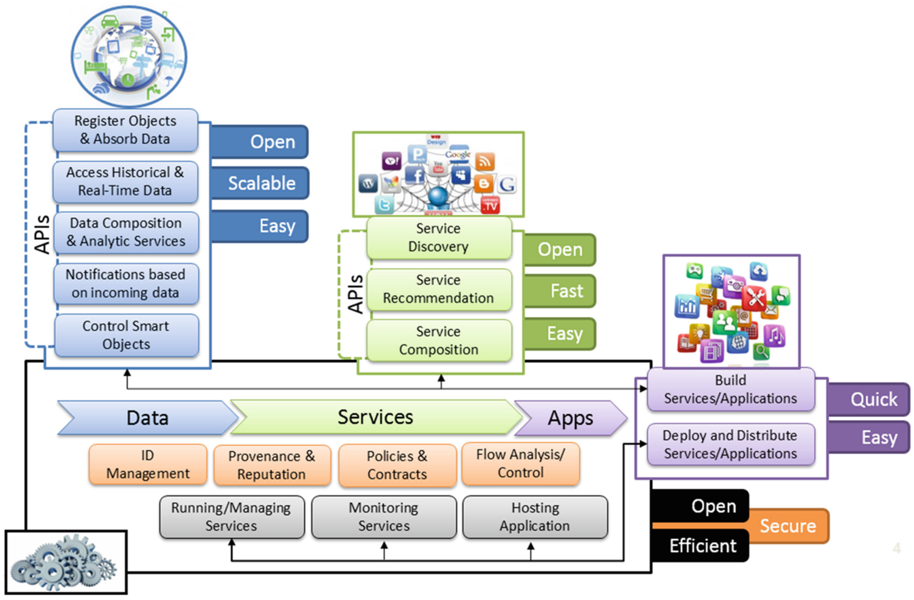

Fig. 1. Conceptional view of the COMPOSE architecture

The third layer, the Ingestion layer, is the interface to the connected objects. Bi-directional communication over different M2M and HTTP-based protocols allows the remote interaction with the devices (e.g., requesting status updates, performing actuations, reading sensor information etc.).

Connected objects play one of the most important roles in IoT systems since the core concept is data aggregation and interaction with smart devices [9]. Within this context, COMPOSE has developed servIoTicy, a data storing and streaming framework with support for device interaction [15].

To interact with real-world objects, users, and services, servIoTicy exposes both, RESTful Application Programming Interfaces (APIs) and M2M protocols (like MQTT, STOMP, and WebSockets). Through these interfaces, devices can store sensor information on the platform. Developers can be notified about updates or retrieve data based on special queries (e.g., time-series based analysis). For this purpose, servIoTicy integrates ElasticSearch [1]. For sensor data stream processing, the Apache Storm [16] component is used, in addition to CouchBase [4] for storing data. Through exposing device communication via REST protocols, servIoTicy also provides an important bridge between REST and MQTT/WebSockets/STOMP [6] as most of the M2M protocols cannot be utilised within a browser.

To simplify the creation and deployment of applications, COMPOSE provides glue.things (http://www.gluethings.com). It is a web-based application to register and access COMPOSE components. It mainly offers the following features: (1) Creating and configuring virtual smart objects in servIoTicy (allowing them to store sensor data, create subscriptions to events, and generate actuations), (2) testing the deployment of sensors by receiving sensor data in real time, and (3) creating IoT applications through a visual workflow editor.

The latter is based on the popular open-source tool Node-RED (http://www.nodered.org). This editor for the IoT allows the easy design of workflows based on components that communicate with devices or services. It is based on node.js and can be executed as a standalone or be integrated in a users application. Workflows (called *flows*) are created in a drag-n-drop fashion by selecting available processes (called *nodes*) that can communicate with external services COMPOSE services.

For the development of applications for connected objects, as well as for mobile or web applications, COMPOSE provides libraries and mobile SDKs as part of glue.things. Libraries for popular embedded platforms (like Arduino, Flyport, mBed, SparkCore, etc.) are provided to simplify the interaction with servIoTicy over the available protocols. In addition, JavaScript libraries are provided for web application development and mobile app development using cross-platform frameworks like the Titanium Appcelerator.

The freedom to interact with numerous devices and applications, the ability to process a new magnitude of data in completely novel ways, and the simplification of the development process comes with a burden: The provisioning of a security framework that supports the openness of IoT, ensures the governance over data, and supports non-security-experts in the development of secure applications.

This burden is particularly hard when considering the application of existing security frameworks. Instead of fixed architectures and pre- and well-defined application scenarios, we face unpredictable contexts in which data is processed and applications are executed. Data becomes easily reusable, may be processed by various types of applications with different functionalities and properties. Further, applications simply emerge from the combination of other services or applications. Their internal complexity may be completely hidden from the developer and their impact on data may be unknown or very hard to determine. Thus, static security perimeters around applications or devices are infeasible and the specification of their security policies is simply impossible.

2 Design Decisions

Our security framework addresses these issues by shrinking the security perimeter to the granularity of data. Instead of forcing developers to foresee which possible application scenarios he wants to cover and which security policies and enforcement technologies are required for that, we ask the entities generating data in the system to define security policies for this very data. This idea is inspired by the mechanisms designed to protect privacy by using the decentralised label model (DLM) [10] extending flow policies introduced by Denning and Denning [5]. Our approach mainly differs in the application of such techniques to highly dynamic architectures in which enforcement must be applied in an ad-hoc fashion and various computational entities. Even more important is the fact that we consider user-defined security policies. Users specify how, by whom, and in which context their data should be processed and which data should gain which kind of access.

Of course, fine-granular policies require complex evaluation and enforcement machineries. This particularly holds for the IoT where additional dimensions, such as spatial or temporal constraints need to be enforceable. Further, policies must not only be able to define the specific security requirements for data and entities but it must also be possible to evaluate them efficiently during static analysis of software and networks as well as during dynamic flow enforcement. Thus, a unified policy framework for our architecture is inevitable. Through the combination of this unified policy framework with an attribute based identity management (IDM) our security framework also simplifies the development of policy enforcement tailored to specific applications.

The IoT will also deploy user-defined code. Thus, devices may run legacy, vulnerable, or manipulated code changing their intentional functionality. They may be physically manipulated to generate new data reading or perform different actions. Further, services which process and consume IoT data may contain malicious or vulnerable code. As a consequence, it is essential that a security architecture for the IoT can detect misbehaving entities which do not allow direct security enforcement. Hence, our architecture provides a reputation system, monitoring devices, and security services. This also allows to use reputation values as policy dimensions.

Finally, our security architecture aims for the definition of novel data security policies by generating provenance information for individual data items. Through a complete history of data, users will be able to define security policies which can also prevent complex data harvesting attacks.

3 Architecture Overview

Centre of the security architecture (see Fig. 2) is the security core. It hosts essential components such as an attribute based IDM and the global policy decision (PDP) as well as policy information points (PIP). While our architecture also supports local instances of these components we use centralised servers to guarantee the consistency of security critical data. The same holds for additional components that extend the functionality of the security core: The reputation- and provenance manager, the static analysis, and the instrumentation component. Reduced to these components, the architecture resembles classical security architectures. Thus, the remainder of this section briefly outlines the functionality of the components relevant for our data-centric framework before going into more detail in the following sections.

To abstract from the complexity of servIoTicy we distinguish three main components: Data Store, Data Management, and Service Object Registry. The latter administrates virtual representations of devices, together with their security policies and reputation information. In particular, reputation information

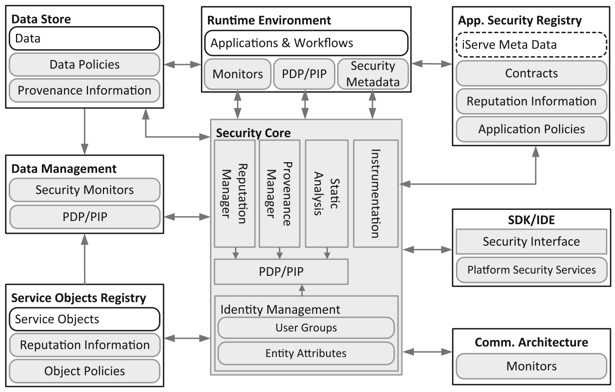

Fig. 2. Outline of the COMPOSE security architecture

is updated frequently according to the use of the associated devices. As soon as devices generate data, it is stored in the Data Store together with its appropriate flow policies derived from device policies. Every time data is used, provenance in the data store is updated based on the operations performed.

The data management which manages access to data and service objects, deploys a local PDP and PIP. In this way, appropriate security monitors can control access within the central data processing component of the overall architecture. The authorised usage of devices and data can be enforced and provenance and reputation information can be generated. Additionally, local security monitors can control data-centric flow-control policies.

servIoTicy can forward data to the runtime environment where it can be processed by applications. They are either provided by COMPOSE or implemented and/or composed by users of the platform. Similar to the data management layer, the execution of every application is secured by dynamic security monitors which can be either holistic, i.e., all components of an application are monitored, or selected in-lined reference monitors. In this way, our architecture enables the fine-granular flow tracking and enforcement of data and the monitoring of specific application properties. For efficient local enforcement and data accumulation, this architectural setup also requires a local PDP, PIP, and stores which allow the local accumulation of security metadata. System-wide security meta data about an application, i.e. reputation information, ownership, or access rules, are stored in the security registry.

The simplification of the creation and deployment of IoT applications is an essential functionality offered by COMPOSE. Hence, it is important to complement this functionality with mechanisms that allow non-experts to assess the compliance of their application with various security requirements. For this purpose, we provide features for editing policy settings, validating and re-configuring data flows within applications, for checking provenance, and for assessing the reputation of system entities.

Finally, to control access from external entities additional security monitors in the communication infrastructure of COMPOSE are deployed.

4 Identity Management

Our platform offers an attribute-based approach. Every user can tag himself or his entities with attribute values. Once entities are tagged with attributes, e.g. the brand of the device, they can be used to specify security policies, e.g. accept data only from devices from brand X. Main problem with this approach is to ensure trust in this information without creating a centralised authority.

Our platform solves this by providing a generic attribute-based IDM framework. This framework allows users to approve attribute information depending on the group where they belong [12]. More specifically, a group membership is defined as a tuple (u, r, g) where user u has role r in group g. But, before a membership is considered effective by the security framework, it has to be approved by two parties: the administrator/owner of g, and u. This mutual agreement policy ensures that users cannot be misplaced in a group against their will, and also that groups contain only users approved by administrators. As a result, users of the platform can rely on groups of their choosing, e.g. the group containing distributors of devices of brand X, to approve attributes used in their policies, e.g. brand of the device.

5 Flow Enforcement

Inputs and outputs of any COMPOSE entity which can process data are annotated with a data-centric flow policy. Apart from specifying the entities allowed to access, execute, or alter a component, flow policies also describe the security requirements of data entering a component and the security properties of data leaving it. Thus, each data item is annotated with security meta-data. Therefore, a unified policy framework is required to avoid additional evaluation overhead. The policy language used in COMPOSE is inspired by ParaLocks [2].

Their main idea is to logically specify with so called *parameterized locks* when a possibly polymorphic actor can retrieve information about a data item. Open locks represent fulfilled conditions under which information can flow. A set of such locks is interpreted as a conjunction of conditions. Combining those conjunctions by disjunctions yield a policy. We adopt this security specification as it is also based on the DLM, it is simple, evaluation is efficient, and it can be used to map against classical access control schema.

To obtain a specification language feasible for the IoT domain, we merge the Usage Control approach $UCON_{ABC}$ [11] with ParaLocks and obtain so called *UsageLocks*. It introduces typed actor and item locks which allow the checking of actor and item attributes, defined by IDM. These locks prevent a blow up of the set of required locks (each attribute could be modeled with another global lock) and they introduce a greater flexibility as pre-defined locks can be used to check user-defined attributes. Further, the new lock types also allow the definition of security contexts which depend on the data items themselves, their usage, and on temporal or spatial constraints. Finally, we also distinguish between flow-to- and flow-from-rules. While the first type of rule maps to the Horn clauses introduced by ParaLocks, flow-from-rules invert this formalism to also allow the specification of rules which describe conditions under which the modification of data or resources is allowed.

To maintain scalability and efficiency, the enforcement mechanisms for the policies described above, require techniques to avoid the dynamic and static analyses and evaluation of policies whenever possible. We partially achieve this through the generation of contracts which are also modeled using the lock-rule system. A static analysis generates an over-approximation of the behaviour of an entity and stores it in a JSON format which uses locks to specify how the programming logic of an entity impacts the flow of information, e.g. by indicating under which conditions (lock status) a flow from an input parameter to an output, such as a file, a socket, or to another COMPOSE entity takes place.

5.1 Dynamic Flow Enforcement

glue.things is based on node.js, i.e. the components used to build an application run on top of node.js and the interaction and flow of data between single nodes is managed by Node-RED. In order to support the enforcement of data-centric security policies even when facing user-defined JavaScript code in applications, we modified the execution environment for single nodes by integrating JSFlow [7]. It is a security-enhanced JavaScript interpreter which allows dynamic flow tracking. It covers the full non-strict JavaScript as specified by the ECMA-262 standard and allows the annotation of values with basic security labels. We extended these labels and the JSFlow infrastructure to support UsageLocks and their policies. Further, we extended JSFlow to support most language features and libraries of node.js and Node-RED.

To also track the information flow between nodes, we modified the basic node template of Node-RED. The primitives used for sending and receiving messages between nodes have been extended. The message exchange now ensures that security information for data, i.e. security policies and lock states, is available in the nodes processing them but protected from the user. To also support basic Node-RED node types, such as function nodes, we further modified the execution primitives to apply our modified JSFlow. To maintain scalability and decrease the performance impact, all other nodes, i.e. node templates deployed by the COMPOSE provider, only symbolically execute the contract of a node to propagate the security labels for data. This removes the need to apply JSFlow to the complete execution of a node but requires additional pre-processing as explained below.

5.2 Static Flow Enforcement

Evidently, contracts are an important pillar for the scalability of our architecture. They are generated by an over-approximating static analysis of the flows generated by an application. The analysis for basic nodes, i.e. nodes which do not represent another application composed, is based on TAJS [8]. Comparable to JSFlow, we extended this static analyser by important language features required to analyse Node-RED applications and to process policies based on UsageLocks. Results from this version of TAJS can be transformed into a contract indicating the flow-relevant behaviour of a COMPOSE entity.

To analyse and generate contracts for complete workflows, a simple model checker has been implemented. It exploits the flow description provided by Node-RED and extends it with contracts for applications which have already been analysed. Basic nodes without contracts are analysed by our TAJS derivate to generate and store their contract. Composed nodes, are forwarded to our model checker. On top of the model resulting from this analysis cascade, the checker identifies non-compliant data flows, i.e. it searches for traces which perform access to data items although locks specified in the security requirements for these items have not been opened.

These analysis results are also used in glue.things. As a consequence, developers with little or no security expertise are able to validate the compliance of applications with the security requirements of their data or the data of other COMPOSE users. This prevents the deployment of insecure applications which will then be subject to dynamic enforcement.

The use of contracts can tremendously simplify the analysis of complex software and support dynamic enforcement. We further exploits the precise information generated during the static analysis to allow code instrumentation. This does not only involve the instrumentation of JavaScript code in user-defined function or customised Node-RED nodes, but it also includes the instrumentation of workflows. Thus, we are able to further reduce the performance impact of the dynamic flow control and can integrate logging or enforcement mechanisms, e.g. prevent access to particular files, in security critical control flows of applications.

6 Reputation

The reputation manager (see Fig. 2), also called PopularIoTy [13,14], aims to cover scenarios when additional information collected during runtime, and users' feedback can help to assess whether a certain application or Service Object is providing a "good" service. Therefore, popularIoTy calculates reputation for entities based on three aspects: popularity, activity, and feedback.

Popularity reflects how often a certain Service Object or application is used, i.e. invoked by other entities. In the case of the data management, whenever data is generated, notifications are stored in the data storage. Likewise, the monitors placed in the runtime store notifications when an application is called. This information is processed to calculate a popularity score.

The *activity* score attempts to reflect whether an entity is behaving properly. In specific cases, data items, i.e. sensor updates, within the data management can be discarded due to several reasons: lack of security policy compliance, developer-provided source code interpretation errors, or developer-intended filtering. When sensor updates are discarded due to the previous reasons, a notification is stored in the data store. Afterwards, this information is used by popularIoTy to decrease the activity score for Service Objects that drop sensor updates, especially in the case of policy compliance and source code problems. To assess whether an application behaves as expected, security contracts are used. Security contracts for applications could be refined by developers when it is not feasible to determine certain flows, e.g. data is sent to a URL received as a parameter. This allows the reputation manager to leverage the monitors placed in the runtime to detect when the application behaves as promised by the developer in the contract refinement. In case the application complies, it will get a positive activity score reward.

PopularIoTy also encourages users to contribute to the reputation calculation through *feedback*. It is comprised of text, and a numerical value reflecting the user's perception about the Service Object or application.

7 Data Provenance

The data provenance manager tracks origins of data, the operations performed on it, and the time when operations took place. This empowers users to define policies based on data provenance, e.g. allow a Service Object to receive data only if it has been processed by a particular application. Further, visualising the provenance of data can help users to detect when certain errors occur; for example, if several data sources are combined, but one of them is malfunctioning, the developer could examine the sources of correct values, and compare them with wrong values to isolate the malfunctioning device. Also, provenance information has an interesting potential to help to protect the user's privacy. For instance, it could eventually help to detect when particular applications harvest and correlate information from specific entities, hinting to the possibility of user profiling.

8 Conclusion

The ability to simplify the development of applications for the IoT and mitigating the overhead for their deployment is essential for the emerging and wide-spread installation of smart devices. COMPOSE provides important tools to support this process and offers a platform for the design and implementation of both innovative and experimental as well as business application scenarios. We have shown how to accommodate this rapid development processes with data-centric security mechanisms. There are far simpler security technologies which could be implemented with less effort and a smaller performance impact. However, they require a clear security expertise at the developers side, concise and complete requirement collections, carefully selected security primitives at specific enforcement points and usually produce isolated silos of devices and services for specific application scenarios. The security paradigm chosen in COMPOSE removes this burden from the developer and delegates it to sophisticated security mechanisms.

They reconfigure and relocate security enforcement as needed and in accordance to the security requirements a user specifies for his data. In this way, our architecture presents the key to open closed silos and supports the most important but seemingly contradictory properties of the IoT: Openness, Simplicity, and Governance.

References

1. Bai, J.: Feasibility analysis of big log data real time search based on Hbase and elasticsearch. In: 9th International Conference on Natural Computation, ICNC 2013, Shenyang, China, pp. 1166–1170, 23–25 July 2013
2. Broberg, N., Sands, D.: Paralocks: role-based information flow control and beyond. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 431–444. ACM, New York (2010)
3. Cloud Foundry (2015). https://storm.incubator.apache.org
4. Couchbase (2015). http://www.couchbase.com/
5. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
6. Doukas, C., Pérez, J.L., Villalba, A., Carrera, D.: Bridging web technologies with M2M platforms. In: W3C Workshop on the Web of Things, Enablers and Services for an Open Web of Devices, Germany, Berlin, pp. 25–26, June 2014
7. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: tracking information flow in JavaScript and its apis. In: Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea, pp. 1663–1671, 24–28 March 2014
8. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
9. Kortuem, G., Kawsar, F., Sundramoorthy, V., Fitton, D.: Smart objects as building blocks for the internet of things. IEEE Internet Comput. **14**(1), 44–51 (2010)
10. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000)
11. Park, J., Sandhu, R.: The UCON$_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. **7**(1), 128–174 (2004)
12. Parra Rodriguez, J.D., Schreckling, D., Posegga, J.: Identity management in platforms offering IoT as a service. In: Jara, A.J., et al. (eds.) IoT 2014. LNICST, vol. 150, pp. 281–288. Springer, Heidelberg (2015). doi:10.1007/978-3-319-19656-5_40
13. Parra, J.D.: Popularioty (2014). http://github.com/nopbyte/popularioty-api/
14. Parra, J.D.: Popularioty Analytics (2014). http://github.com/nopbyte/popularioty-analytics/
15. servIoTicy: IoT streaming made easy (2015). http://www.servioticy.com/
16. The Apache Storm distributed real-time communication system (2015). https://storm.incubator.apache.org/