Security Implications of Permission Models in Smart-Home Application Frameworks

Earlence Fernandes and Amir Rahmati | University of Michigan
Jaeyeon Jung | Samsung
Atul Prakash | University of Michigan

Analysis of a popular programming framework reveals that many smart-home apps are automatically overprivileged, leaving users at risk for remote attacks that can cause physical, financial, and psychological harm.

Smart-home technology has evolved beyond basic convenience functionality, such as automatically controlled lights and door openers, to provide tangible benefits. For instance, water flow sensors and smart meters facilitate energy efficiency. IP-enabled cameras, motion sensors, and connected door locks offer better control of home security. However, attackers can manipulate smart devices to cause users physical, financial, and psychological harm. For example, burglars can target a connected door lock to plant hidden access codes.[1]

Early smart-home systems had steep learning curves and complicated device setup procedures and thus were limited to do-it-yourself enthusiasts. (Many forums exist for people to exchange know-how, such as forum .universal-devices.com.) Recently, several companies introduced cloud-backed systems that are easier for users to set up and that provide a programming framework for third-party developers to build smart-home apps. Examples of such frameworks are Samsung's SmartThings (www.smartthings.com), Apple's HomeKit (www.apple.com/ios/home), Vera Control's Vera3 (getvera.com/controllers/vera3), Google's Weave/Brillo (developers.google.com/weave), and AllSeen Alliance's AllJoyn (including Qualcomm, Microsoft, LG, Cisco, and AT&T; allseenalliance.org/framework).

We consider the security implications of a key component of such smart-home programming frameworks: their permission models. These models limit the risk third-party apps pose to users and their devices. We first survey the permission models of Apple HomeKit, IoTivity, AllJoyn, and SmartThings, then discuss results from a deep-dive analysis of the SmartThings framework.[2]

Smart-Home Permission Models

In our survey of existing permission models of upcoming smart-home frameworks, we observed varying granularity in access control, ranging from all-or-nothing to very fine grained.

HomeKit is a framework and set of protocols that enable smart-home devices to communicate securely with iOS devices and apps. Third- and first-party developers can write apps for HomeKit Accessories. A first-party app is written by a device manufacturer.

HomeKit represents a physical device as an `HMAccessory` object that exposes objects of type `HMService`. For example, an accessory might be a

 Copublished by the IEEE Computer and Reliability Societies

garage opener, and it might have the following services: light and switch. Services can have characteristics that apps manipulate to cause physical changes in the accessory's or device's state. For example, the switch service might have an on/off characteristic that results in unlocking and locking the garage opener. iOS apps gain access to accessories at the home granularity. That is, either an app has access to all accessories in the home, or to none at all. Users must grant apps access to HomeKit data, similar to the iOS experience of granting apps access to data like contacts and photos.[3] Home-level granularity implies that all apps are overprivileged automatically.

> **Our key finding is that overprivilege is a significant shortcoming of the SmartThings permission model.**

IoTivity is an open source framework sponsored by the Open Connectivity Foundation (OCF), which includes Microsoft, Intel, Samsung, and Qualcomm.[4] IoTivity's goal is to create an open source reference implementation of OCF Internet of Things (IoT) standards to facilitate communication of IoT devices with one another and the Internet.

IoTivity doesn't include security features by default. To enable these functions, the `SECURED=1` flag has to be set during compilation. When this flag is turned on, an IoTivity server hosting a resource (that is, a device) can impose access control on it by assigning the `OC_SECURE` property to it during creation. This access control is limited to coarse-grained write and read permissions associated with the ID of devices that want to communicate with the resource. These permissions are set when a device is added and can later be modified by the user. To add a device to its network when security features are on, IoTivity supports three protocols:

- *just work*, in which a shared key is established during first communication;
- *random PIN*, in which an off-band PIN is required for establishing trust; and
- *asymmetric key*, which is based on a self-signed or manufacturer key.

In secure mode, datagram TLS (DTLS) protects communication between secured resources.

AllJoyn is an open standard that enables various physical devices and apps to communicate in a uniform way. It consists of a communication protocol and a software library that device builders and app developers must use. The software library runs on a variety of hardware. AllJoyn has a distributed architecture with no central controller or hub. It relies on public-key cryptography to secure communications and express access control policies.

A software app or physical device is collectively referred to as an *app* in AllJoyn terminology. An app can expose interfaces that have members. For example, a lock can provide the `control` interface with the members `lock` and `unlock`. Apps can consume interfaces from other apps. For example, an auto-lock app will consume the door lock's `control` interface. AllJoyn standardizes some interface definitions for a select set of devices, such as lights and HVAC.

Apps are security principals and are associated with an identity certificate signed by a certificate authority that all apps must trust. The AllJoyn security manager is a component that speaks the AllJoyn protocol and issues identity certificates to apps. An administrative user, such as a home or building owner, operates the security manager component.

AllJoyn offers arbitrarily granular access control down to the member level. However, recommended access control is at the interface level. Therefore, each AllJoyn app (pure software or physical device) must create a manifest template that lists the set of interfaces the app will provide and the set of interfaces the app will consume. This manifest template represents an app's permission request, similar to how smartphone apps request permission to sensitive resources. During installation, the admin uses the security manager to create a final manifest for the app and then includes a digest of the final manifest in an identity certificate. This step is conceptually similar to a smartphone owner accepting the set of permissions an app requests. The security manager then installs this manifest and identity certificate in the target app.

At runtime when a consumer app wants to invoke an interface on a provider app, the provider app will read the identity certificate, verify it (and the corresponding certificate chain), verify the manifest digest, and then eventually check whether the consumer is allowed to access the provider's interface.

SmartThings provides a hub and cloud back end. Third-party developers write SmartApps that execute in the cloud back end. The SmartThings framework must ensure that SmartApps have only the required privilege to complete their claimed functionality. Therefore, SmartThings has a security architecture—the SmartThings capability model—that governs which devices a

SmartApp might access. A capability is composed of a set of commands (method calls) and attributes (properties). Commands represent the ways in which a device can be controlled or actuated. Attributes represent a device's state information. Table 1 lists example capabilities. A single device can expose a set of capabilities. For example, a smart lock might expose `capability.lock` and `capability.battery`.

SmartApps must request capabilities from the user. When a user installs a SmartApp, the requested capabilities trigger a device enumeration process that scans all the physical devices currently paired with the user's hub, and for each capability request, the user is presented with all devices that support the specified capability. Once the user selects a particular device exposing the specified capability, the SmartApp gains access to that device. SmartApps can interact with devices by using events. In particular, a SmartApp can register a callback on a device for a condition, and whenever that condition becomes true, the SmartApp is given a callback with some optional event data. These apps provide a wide variety of functionality ranging from simple rule-based automation ("if my door lock is open, then turn on the lights") to energy monitoring and saving solutions.

Why SmartThings?

We chose to analyze SmartThings in depth for several reasons. First, it's a relatively mature platform with a growing set of apps—called SmartApps—and it supports 132 types of devices.

Second, SmartThings shares key security design principles with other frameworks. Authorization and authentication for device access are essential in securing smart-home app platforms, and SmartThings has a built-in mechanism to protect device operations against third-party apps through so-called *capabilities*.

Event-driven processing is common in smart-home applications,[5] and SmartThings allows apps to register callbacks for a given event stream generated by a device. Other platforms support event-driven processing too. For instance, AllJoyn supports the bus signal,[6] and HomeKit provides the characteristic notification API.[7] Therefore, we believe lessons learned from an analysis of the SmartThings permission model will inform the early design stages of other programmable smart-home frameworks.

Our key finding is that *overprivilege* is a significant shortcoming of the SmartThings permission model. In particular, we found that SmartApps in our dataset of 499 apps were significantly overprivileged: 55 percent didn't use all the rights to device operations that their requested capabilities implied, and 42 percent were granted capabilities that weren't explicitly requested or used. In many cases, overprivilege was unavoidable as a result of the capability model's device-level authorization design, and occurred through no fault of the developer. Worryingly, we observed that 68 existing SmartApps use overprivilege to provide extra features without requesting the relevant capabilities.

We built attacks that use overprivilege to demonstrate its negative effects. Our attacks can inject PIN codes into a connected door lock, snoop on PIN codes, disable vacation mode, and cause fake fire alarms.

These attacks underline the need for careful permission model design as well as for stronger control over how apps use sensitive data. However, this is often challenging to achieve. The well-known tension between usability and granularity of permission models manifests itself here as well. However, the lessons we've learned about how incorrectly designed models lead to security failures in smart homes, coupled with progress in permission model design in smartphones and other closely related spaces, can lead to improved security.

SmartThings Deep-Dive Analysis

We investigated the security of the SmartThings permission model along two dimensions: least privilege and sensitive-event data protection. After studying the permission model and extensively testing prototype SmartApps, we created a list of potential security issues.

Least-Privilege Principle Adherence

Does the capability model protect sensitive operations of devices against untrusted or benign-but-buggy SmartApps? It's important to ensure that SmartApps request only the privileges they need and are granted only the privileges they request. However, we found that many existing SmartApps are overprivileged.

Sensitive-Event Data Protection

Which access control methods are provided to protect sensitive-event data generated by devices against untrusted or benign-but-buggy SmartApps? We found that unauthorized SmartApps can eavesdrop on sensitive events.

Occurrence of Overprivilege in SmartApps

We found two significant issues with overprivilege in the SmartThings framework, both artifacts of the way its capabilities are designed and enforced. First, capabilities in the SmartThings framework are coarse grained, providing access to multiple commands and attributes for a device. Thus, a SmartApp could acquire the rights to invoke commands on devices even if it doesn't use them. Second, a SmartApp can end up obtaining more capabilities than it requests because of the way the SmartThings framework binds the SmartApp to devices. We detail both issues below.

Table 1. Examples of capabilities in the SmartThings framework.

Capability	Command	Attribute
`capability.lock`	`lock()`, `unlock()`	`lock` (lock status)
`capability.battery`	N/A	`battery` (battery status)
`capability.switch`	`on()`, `off()`	`switch` (switch status)
`capability.alarm`	`off()`, `strobe()`, `siren()`, `both()`	`alarm` (alarm status)
`capability.refresh`	`refresh()`	N/A

Coarse-grained capabilities. In the SmartThings framework, a capability defines a set of commands and attributes. Here is a small example of `capability.lock`:

- Associated commands: `lock` and `unlock`.
- Associated attribute(s): `lock`. The `lock` attribute has the same name as the command, but the attribute refers to the locked or unlocked device status.

Our investigation of the existing capabilities defined in the SmartThings architecture shows that many capabilities are too coarse grained. For example, the auto-lock SmartApp, available in the SmartThings app store, requires only the `lock` command of `capability.lock` but also gets access to the `unlock` command, thus increasing the attack surface if the SmartApp is exploited. If the `lock` command is misused, the SmartApp could lock out authorized household members, causing inconvenience; however, if the `unlock` command is misused, the SmartApp could leave the house vulnerable to break-ins. There's often an asymmetry in risk with device commands. For example, turning on an oven could be dangerous, while turning it off is relatively safe. Thus, it's inappropriate to automatically grant a SmartApp access to an unsafe command when it only needs access to a safe command.

To provide a simple measure of overprivilege due to coarse-grained capabilities, we computed the following for each evaluated SmartApp, based on static analysis and manual inspection: {requested commands and attributes} – {used commands and attributes}. Ideally, this set would be empty for most apps. However, in our analysis of 499 SmartApps, we found that 276 apps are overprivileged due to coarse-grained capabilities.

Coarse SmartApp–SmartDevice binding. When a user installs a SmartApp, the SmartThings platform enumerates all physical devices that support the capabilities declared in the app's `preferences` section, and the user chooses the set of devices to be authorized to the SmartApp. Unfortunately, the user isn't told about the capabilities being requested and is presented only with a list of devices that are compatible with at least one of the requested capabilities. Moreover, once the user selects the devices to be authorized for use by the SmartApp, the SmartApp gains access to all commands and attributes of all the capabilities implemented by the selected devices' handlers. We found that developers couldn't avoid this overprivilege, because it was a consequence of SmartThings framework design.

More concretely, SmartDevices provide access to the corresponding physical devices. Besides managing the physical device and understanding the lower-level protocols, each SmartDevice also exposes a set of capabilities appropriate to the device it manages. For example, the default Z-Wave lock SmartDevice supports the following capabilities: `capability.actuator`, `capability.lock`, `capability.polling`, `capability.refresh`, `capability.sensor`, `capability.lockCodes`, and `capability.battery`.

These capabilities reflect various facets of the lock device's operations. Consider a case in which a SmartApp requests the `capability.battery`, say, to monitor the condition of the lock's battery. The SmartThings framework would ask the user to authorize access to the Z-Wave lock device (because it matches the requested capability). Unfortunately, if the user grants the authorization request, the SmartApp also gains access to the requested capability and all the other capabilities defined for the Z-Wave lock. In particular, the SmartApp would be able to lock and unlock the Z-Wave lock, read its status, and set lock codes.

To provide a simple measure of overprivilege due to unnecessary capabilities being granted, we computed the following for each evaluated SmartApp, based on static analysis and manual inspection: {granted capabilities} – {used capabilities}. Ideally, this set would be empty. However, our analysis found that 213 of the 499 SmartApps were overprivileged due to additional capabilities being granted.

Insufficient Sensitive-Event Data Protection

SmartThings supports a callback pattern in which a SmartDevice can fire events filled with arbitrary data, and SmartApps can register for those events. Inside a

user's home, each SmartDevice is assigned a 128-bit device identifier when it's paired with a hub. After that, a device identifier is stable until it's removed from the hub or paired again. The 128-bit device identifiers are thus unique to a user's home, which is good—possession of the 128-bit device identifier from one home isn't useful in another home. Nevertheless, we found significant vulnerabilities in the way access to events is controlled:

- Once a SmartApp is approved for access to a SmartDevice after a capability request, the SmartApp can monitor any event data published by that SmartDevice. The SmartThings framework has no special mechanism for SmartDevices to selectively send event data to a subset of SmartApps or for users to limit a SmartApp's access to only a subset of events.
- Once a SmartApp acquires the 128-bit identifier for a SmartDevice, it can monitor all the events of that SmartDevice, without gaining any of the capabilities that device supports.
- Certain events can be spoofed. In particular, we found that any SmartApp or SmartDevice can spoof location-related and device-specific events.

Event leakage via capability-based access. As noted earlier, once a user approves a SmartApp's request to access a SmartDevice for any supported capability, the SmartThings framework permits the SmartApp to subscribe to all the SmartDevice's events. We found that SmartDevices extensively use events to communicate sensitive data. For instance, we found that the SmartThings-provided Z-Wave lock SmartDevice transmits `codeReport` events that include lock PIN codes.

A SmartApp with any form of access to the Z-Wave lock SmartDevice (say, for monitoring the device's battery status) can also automatically monitor all its events and use that access to log the events to a remote server and steal lock PIN codes. The SmartApp can also track lock codes as they're being used to enter and exit the premises, therefore tracking household members' movements and possibly violating their privacy.

Event leakage via SmartDevice identifier. As discussed, each SmartDevice in a user's home is assigned a random 128-bit identifier. This identifier, however, isn't hidden from the SmartApps. Once a SmartApp is authorized to communicate with a SmartDevice, it can read the `device.id` value to retrieve the 128-bit SmartDevice identifier. We found that a malicious SmartApp can directly use this identifier to read any events a device generates, irrespective of any granted capabilities.

Unfortunately, the device identifiers are easy to exchange among SmartApps—they aren't opaque handles, nor specific to a single SmartApp. Several SmartApps currently exist in the SmartThings app store that allow remote retrieval of the device identifiers in a user's home over the OAuth protocol.

Event spoofing. The SmartThings framework neither enforces access control around raising events nor offers a way for triggered SmartApps to verify an event's integrity or origin. We discovered that an unprivileged SmartApp can spoof both physical-device and location-related events.

A SmartDevice detects physical changes in a device and raises the appropriate event. For example, a smoke detector SmartDevice will raise the "smoke" event when it detects smoke in its vicinity. The event object contains various state information plus a location identifier, a hub identifier, and the 128-bit device identifier that's the event source. We found that an attacker can create a legitimate event object with the correct identifiers and place arbitrary state information. When such an event is raised, SmartThings propagates the event to all subscribed SmartApps as if the SmartDevice itself triggered the event. Obtaining the identifiers is easy—the hub and location ID are automatically available to all SmartApps.

To summarize, we found that the SmartThings event subsystem design is insecure. SmartDevices extensively use it to post their status and sensitive data—111 out of 132 device handlers from our dataset of device handler code raise events.[2]

Proof-of-Concept Attacks

Using four concrete attacks, we demonstrated how the overprivilege design issue weakens home security. We combined overprivilege with other security design flaws in the SmartThings framework to make the attacks remote and stealthy. For more details on the other design vulnerabilities, see "Security Analysis of Emerging Smart Home Applications."[2]

Table 2 summarizes the four attacks based on insecure design of the permission model in SmartThings. The backdoor PIN code injection attack uses coarse SmartApp–SmartDevice binding overprivilege to force an existing SmartApp to program a PIN code into a door lock. The overprivilege enables the attacker to inject a PIN code programming command into the SmartApp. The door lock PIN code snooping attack is a stealthy malware app that uses overprivilege in the event system of SmartThings to snoop on PIN codes as they're created and then leak them out. Attacks can disable vacation mode by using the lack of access control around the location object to trick a SmartApp into thinking that the home is occupied. Finally, the fake fire alarm attack uses a malware app that escalates its privileges by

Table 2. Four proof-of-concept attacks on SmartThings.

Attack description	Attack vectors	Physical world impact[1]
Backdoor PIN code injection	▪ Command injection to an existing web service SmartApp ▪ Overprivilege using SmartApp–SmartDevice coarse binding ▪ Stealing an OAuth token using the hard-coded secret in the existing binary ▪ Getting a victim to click a link pointing to the SmartThings website	▪ Enabling physical entry ▪ Physical theft
Door lock PIN code snooping	▪ Stealthy attack app that requests only the capability to monitor battery levels of connected devices and gets a victim to install the attack app ▪ Eavesdropping on events data ▪ Overprivilege using SmartApp–SmartDevice coarse binding ▪ Leaking sensitive data using unrestricted short message services	▪ Enabling physical entry ▪ Physical theft
Disabling vacation mode	▪ Attack app with no specific capabilities ▪ Getting a victim to install the attack app ▪ Misusing logic of a benign SmartApp ▪ Event spoofing	▪ Physical theft ▪ Vandalism
False fire alarm	▪ Attack app with no specific capabilities ▪ Getting a victim to install the attack app ▪ Spoofing physical-device events ▪ Controlling devices without gaining appropriate capability ▪ Misusing logic of benign SmartApp	▪ Misinformation ▪ Annoyance

stealing a device identifier and generating fake carbon monoxide sensor readings.

The IoT is predicted to reach 20.8 billion connected devices by 2020, with the consumer sector having the largest installed base.[8] Simultaneously, we're observing the emergence of programmable frameworks that unify disparate devices into a coherent platform that supports third-party app development. Although these third-party apps represent the benefits of networked and intelligent devices, they also represent the risk that such technologies pose. In this article, we surveyed the permission models of four recent frameworks (IoTivity, HomeKit, AllJoyn, and SmartThings) because a permission model is the first line of defense between the users' privacy-sensitive data and physical devices, and attackers. A permission model with security design deficiencies will lead to various kinds of attacks. Our findings underline the need for more research on permission models and how apps use data once they gain access.[9]

Our SmartThings analysis prompted SmartThings developers to begin designing and implementing techniques to reduce automatic overprivilege and better balance capability granularity and usability based on our ideas.[2] As a first line of defense, the SmartThings team revised its app review guidelines to manually look for use and misuse of overprivilege based on the attacks we created. As defense in depth, SmartThings also revised its developer documentation to discuss security best practices. For example, it instructs developers to be precise in the kinds of events they subscribe to, and to not use Groovy dynamic method execution unless it's explicitly guarded by statements that ensure no unintended actions are performed, thus preventing remote attackers from exploiting overprivilege.[10] ■

References

1. T. Denning, T. Kohno, and H.M. Levy, "Computer Security and the Modern Home," *Comm. ACM*, vol. 56, no. 1, 2013, pp. 94–103.
2. E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," *Proc. 37th IEEE Symp. Security and Privacy*, 2016; dx.doi.org/10.1109/SP.2016.44.
3. "iOS Security Guide," Apple, May 2016; www.apple.com/business/docs/iOS_Security_Guide.pdf.
4. "Open Connectivity Foundation," IoTivity, 2016; www.iotivity.org.

5. B. Ur et al., "Practical Trigger-Action Programming in the Smart Home," *Proc. SIGCHI Conf. Human Factors in Computing Systems* (CHI 14), 2014, pp. 803–812.
6. "Documentation: Data Exchange," AllSeen Alliance; allseenalliance.org/framework/documentation/learn/core/system-description/data-exchange.
7. "HMCharacteristic," Apple, 2017; https://developer.apple.com/reference/homekit/hmcharacteristic.
8. "Gartner Says 6.4 Billion Connected 'Things' Will Be in Use in 2016, Up 30 Percent from 2015," Gartner, 10 Nov. 2015; www.gartner.com/newsroom/id/3165317.
9. E. Fernandes et al., "Flowfence: Practical Data Protection for Emerging IoT Application Frameworks," *Proc. 25th USENIX Security Symp.* (USENIX Security 16), 2016, pp. 531–548.
10. "Code Review Guidelines and Best Practices: Security Considerations," SmartThings Developer Documentation, 2016; docs.smartthings.com/en/latest/code-review-guidelines.html#security-considerations.

Earlence Fernandes is a PhD candidate at the University of Michigan. His research focuses on techniques that enable secure and safe Internet of Things (IoT) platforms, including building secure systems, finding design vulnerabilities, and conducting large-scale measurements. Contact him at earlence@umich.edu.

Amir Rahmati is a PhD candidate at the University of Michigan. His research focuses on improving the security of emerging technologies and resource limited devices, such as embedded and IoT devices. Rahmati received an MSE in computer science and engineering from the University of Michigan. Contact him at rahmati@umich.edu.

Jaeyeon Jung is vice president of Samsung's Cloud Platform group. Her research focuses on developing new technologies for protecting consumer privacy, particularly in the areas of mobile systems and emerging connected devices for the home. Contact her at jae.jung@samsung.com.

Atul Prakash is a professor of computer science at the University of Michigan. His research spans security and privacy, cyber-physical systems, computer-supported cooperative work, and distributed systems. Contact him at aprakash@umich.edu.