Security Analysis of Emerging Smart Home Applications

Earlence Fernandes · Jaeyeon Jung · Atul Prakash

IEEE S&P · OAKLAND '16

The 2016 IoT App Landscape

SmartThings: 521 apps

Vera: 204 apps

HomeKit · Weave/Brillo · AllJoyn: <50 each

100–500k

Android companion app installs, March 2016

PRIOR RESEARCH

Examined individual devices and local protocols, not the shared programming framework connecting them.

Android IPC/Intent Security

Decoupled, asynchronous messaging drives action, but doesn't check provenance

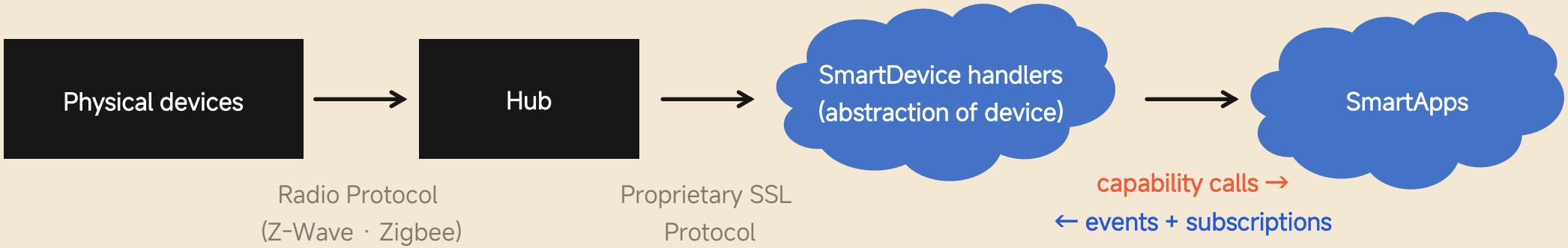
OAuth Confusion

Mobile dev misunderstanding lead to hard-coded secret that enables the backdoor PIN PoC

Hub/Infrastructure Audit

Examined hubs including SmartThings for SSL/TLS, replay protection, password strength.

The SmartThings Abstraction



- Apps declare **capabilities** (battery, lock, switch); installation binds matching physical devices to those requests.
- Capabilities group commands and attributes behind one interface; **events + subscriptions** drive trigger-action automation.
- Capabilities and events let apps control devices without touching protocols.

IMPLEMENTATION DEFENSES

Groovy sandboxing · OAuth for third-party integrations · Secure storage · Encrypted component links

The failures ahead live in the design of the authorization and event layers around these defenses.

Does the capability model enforce least privilege?

Permission Units Don't Match Risk

Asymmetric Risk

Capability	Commands / attributes
capability.battery	battery level
capability.lock	lock() · unlock()

- **oven.on()** can start a fire; **oven.off()** cannot
- Locking when owner is out is safe; locking while inside might not be
- Unlocking your door is safe, unlocking a stranger's door is not
    - All sit inside capability.lock.

Even Worse…

Selecting one device authorizes the app to **every capability** in the handler.

The install screen asked users to pick devices **without revealing the full capability set** being granted (no visible/granular permissions).

Least privilege fails when the unit of authorization ignores context and consequences.

Analyzing a Closed Platform from the Outside

A (somewhat) classic security workflow:

01
Collect source
Apps + handlers pulled via discovered backend endpoints; 22 binary-only apps unobtainable.

02
Recover definitions
Device handlers queried against a discovered REST endpoint; returned capabilities w/ commands/attributes.

03
AST static analysis
Groovy compilation customizer; reflection defeats bytecode tools such as Soot.

04
Manual validation
Dynamic-invocation cases reviewed by hand; physical tests on real hardware.

SmartThings published incomplete docs:

66 → 93

Commands documented -> commands recovered

60 → 85

Attributes documented -> attributes recovered

4.8%

false-positive rate for static analysis:
24 of 499 apps misclassified as overprivileged

499 source-available SmartApps / 132 device handlers

Overprivilege Was Structural, Not Incidental

55%

276 / 499 received unused commands or attributes via coarse capabilities

43%

213 / 499 received extra, unused capabilities via device binding

Does the capability model enforce least privilege?

Permission Units Don't Match Risk

Asymmetric Risk

Capability	Commands / attributes
capability.battery	battery level
capability.lock	lock() · unlock()

- **oven.on()** can start a fire; **oven.off()** cannot
- Locking when owner is out is safe; locking while inside might not be
- Unlocking your door is safe, unlocking a stranger's door is not
    - All sit inside capability.lock.

Even Worse…

Selecting one device authorizes the app to **every capability** in the handler.

The install screen asked users to pick devices **without revealing the full capability set** being granted (no visible/granular permissions).

Least privilege fails when the unit of authorization ignores context and consequences.

Overprivilege Was Structural, Not Incidental

55%

276 / 499 received unused commands or attributes via coarse capabilities

43%

213 / 499 received extra, unused capabilities via device binding

13.6%

68 / 499 used capabilities they never requested + compatibility pressure to keep the flaw

Surrounding Attack Surface

111 / 132

device handlers raise events

131

apps use unrestricted SMS APIs

36

use unrestricted Internet APIs

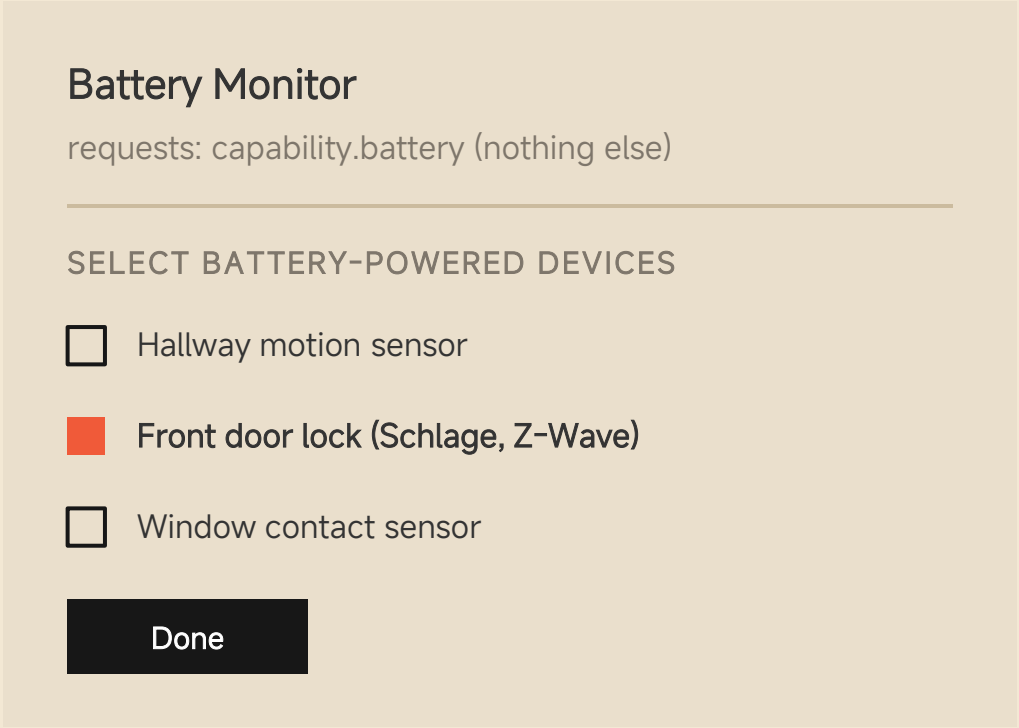
26

use dynamic method invocation

27

are OAuth-enabled

“Monitor My Batteries”



Install UI recreated after Fig. 2 (i.e., every compatible device is offered)

Even Worse…

Selecting one device authorizes the app to **every capability** in the handler.

The install screen asked users to pick devices **without revealing the full capability set** being granted (no visible/granular permissions).

The user sees devices. The framework grants capabilities.

Capabilities Granted

REQUESTED

capability.battery

Nothing else requested (by the developer or the user)

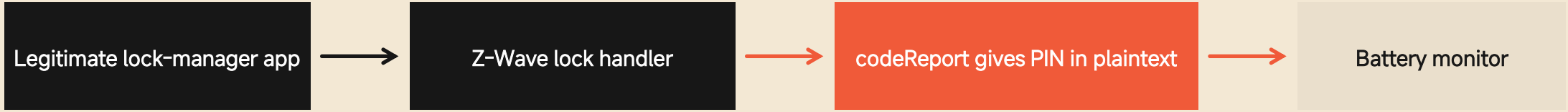
RECEIVED: FULL Z-WAVE LOCK HANDLER

actuator · lock · polling · refresh

sensor · lockCodes · **battery**

Includes unlock(), setCode(), and codeReport PIN events.

PIN Snooping Through the Event Path



OBSERVED EVENT PAYLOAD

```
{ "name": "codeReport", "value": 4,
"data": { "code": "8877" },
"descriptionText": "…Schlage Lock code 4 set" }
```

Tested on real infrastructure: **Schlage FE599** lock, smart outlet, SmartThings hub, store-obtained lock manager.

Asked for battery status. Received the keys to the door.

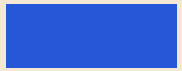
User Approval

Survey of 22 SmartThings owners (Table VI)

Would let the battery monitor watch the **Schlage lock**

91% 20/22

Thought it could send **door-access codes** to a remote server



14% 3/22

Would be **upset** if it actually did

100% 22/22

STRENGTHENS THE POINT

12 / 22 had 6+ years of professional programming experience.

17 / 22 were interested or very interested in the battery monitor itself.

Backdoor PIN Injection (No Malicious App)

01

Steal the OAuth token

A popular Android companion app **hard-coded its client ID + secret** in bytecode.

The attacker links to the **genuine** SmartThings HTTPS login page with only redirect_uri

02

Inject the command

An existing WebService SmartApp executes HTTP-supplied strings via Groovy dynamic injection*

***26** apps used dynamic invocation; **20** in WebService handlers where a remote party supplies the string.

03

Land on the lock

The injected **setCode** succeeds because the vulnerable app already holds capability.lockCodes.

The planted PIN survives patching the Android app or the hub going offline.

Weaponizing Benign Apps with Fake Events

one line of attack code · zero capabilities held

ATTACK 3 · VACATION-MODE DISABLING

Vacation mode simulates occupancy to deter burglars; an existing SmartApp starts and stops it from the location object's “mode”.

A no-capability app calls **sendLocationEvent** with a false mode change — the benign app stops the simulation itself.

ATTACK 4 · FAKE ALARM

A store alarm-panel app is legitimately authorized for CO detectors and sirens.

A no-capability app **spoofs a CO-detector event**; the alarm panel trusts it and activates the siren.

Compositional Pattern

Attack	Key vector	Physical impact
1 · Backdoor PIN injection	Stolen OAuth token + unsafe dynamic invocation + device-binding overprivilege	Planted lock code
2 · PIN snooping	Disguised battery monitor + event leakage + unrestricted egress	Stolen lock codes
3 · Vacation-mode disabling	Spoofed location event + trusting benign app	Occupancy simulation off
4 · Fake alarm	Spoofed CO event + trusting benign app	Siren sounds

framework flaw + ordinary bug + available egress + physical actuator

Aftermath

Dec 17, 2015
Disclosed to SmartThings

Jan 12, 2016
Response: stronger OAuth tokens promised by April

Apr 15, 2016
Follow-up: manual app-review team: mitigation intent, not verified remediation

"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."

- The concrete mitigation is manual app review aimed at the backdoor-PIN vector (exposed HTTP endpoints and dynamic invocation).
- The message doesn't say whether the promised OAuth-token strengthening actually shipped,
- **AND it claims no structural change to event leakage, event spoofing, or the capability model.**

SmartThings seems to lean on manual review, **but the demo battery monitor was built to look benign in source and fetched its malicious behavior remotely**.

Aftermath

Apr 29, 2016

Vulnerability re-tested, confirmed active[1]

May 2, 2016

Author call with SmartThings to discuss redesign: “frameworks are hard to change after adoption” [2]

Sep 30, 2022

Groovy SmartApps deprecated and IDE removed, replaced by local Lua Edge drivers/API-first integrations[3]

Follow-up Work:

- **SmartAuth** (Tian et al., USENIX Security 2017)
- **ContextIoT** (Jia et al., NDSS 2017)
- **IoTGuard** (Celik et al., NDSS 2019)
- **Soteria** (Celik et al., USENIX ATC 2018)

[1] https://www.scworld.com/news/university-of-michigan-researchers-remotely-pick-locks-of-samsung-smartthings-connected-home-systems
[2] https://www.ieee-security.org/TC/SP2016/slides/24-4/fernandes.pdf
[3] https://support.smartthings.com/hc/en-us/articles/9339624925204

Three Lessons from the Authors

01 Risk-based capabilities

Split functionally-paired but risk-asymmetric operations; adapt smartphone-permission user-study methods with users, manufacturers, and platforms. Accept the usability cost of finer granularity.

02 Event identity + selective dissemination

Authenticate event origin, verify integrity, control who may raise events, deliver sensitive events selectively. **Identity is necessary but not sufficient. Receivers also have to *check* provenance, and platforms need access control around who may raise what.**

03 Cooperating, vetting app stores

A smart-home store and a mobile store each see only half of an integration: the hard-coded-secret Android app passed Google's vetting because the Groovy side was invisible to it.

Ecosystem Responsibility

Platform providers: define capabilities, event identity, runtime restrictions, disclosure interfaces.

App stores, smart-home + mobile: each sees half of an integration; need cooperative vetting signals (the OAuth attack).

Companion-app developers: handle OAuth secrets, redirects, external input, remote control paths.

Device vendors: define handler behavior, sensitive event contents, firmware lifecycles, replacement policies.

Consumers: make installation and maintenance choices but cannot(?) evaluate authority the ecosystem hides.

Discussion: Permissions and Provenance

Q1

Risk-based permissions can cut overprivilege but add prompts and configuration. **What permission granularity would you accept in your own home, and who should decide which IoT operations are dangerous?**

Discussion: The Manufacturer as Adversary

Q2

If a manufacturer intentionally **denies** access to a purchased IoT device until its owner makes another **should consumers treat that manufacturer as an adversary?**

How much responsibility should consumers bear for anticipating such behavior before purchase (even when the authority at stake was never shown to them)?

Q3

In 2026, we see the exact same pattern of over-privilege, unverified triggers, and unrestricted egress in LLM agents.

Why haven't we learned anything in the last 10-250 years?