CIS 4930: Secure IoT

Prof. Kaushal Kafle

Lecture 10

Platforms

SmartThings
(pre-2019)

SmartThings

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

Google Nest

nest

Philips Hue

PHILIPS

hue

A Study of Data Store-based Home Automation

Kaushal Kafle, Kevin Moran, Sunil Manandhar, Adwait Nadkarni, Denys Poshyvanyk
William & Mary, Williamsburg, VA, USA
{kkafle,kpmoran,smanandhar,nadkarni,denys}@cs.wm.edu

Background: SmartThings

Capabilities = *Commands* ⊕ *Attributes*

e.g. on( ), off( ) — *e.g. switch, battery*

EXAMPLES OF CAPABILITIES IN THE SMARTTHINGS FRAMEWORK

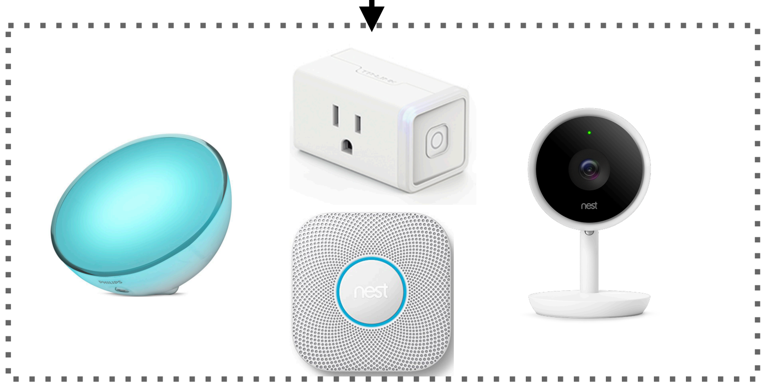
Capability	Commands	Attributes
`capability.lock`	`lock()`, `unlock()`	`lock` (lock status)
`capability.battery`	N/A	`battery` (battery status)
`capability.switch`	`on()`, `off()`	`switch` (switch status)
`capability.alarm`	`off()`, `strobe()`, `siren()`, `both()`	`alarm` (alarm status)
`capability.refresh`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

Background: SmartThings

SmartApps Mini-apps written to facilitate trigger-action programming

- Written using the SmartThings Developer SDK
- Language Groovy, compiles to Java byte code
- Execute in the SmartThings cloud backend (closed-source)

Device Handlers Software-wrappers for physical devices



Background: SmartThings

SmartApps Mini-apps written to facilitate trigger-action programming

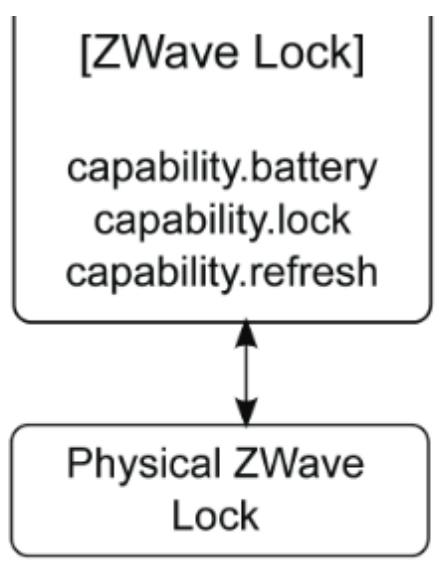
1. Device Handlers declare a device's capability.
2. SmartApps request devices with specific capabilities.
3. Users *bind* SmartApps to devices through Device Handlers.

2

```
//query the user for capabilities
preferences {
  section("Select Devices") {
    input "lock1", "capability.lock", title:
        "Select a lock"
    input "sw1", "capability.switch", title:
        "Select a switch"
  }
}
```

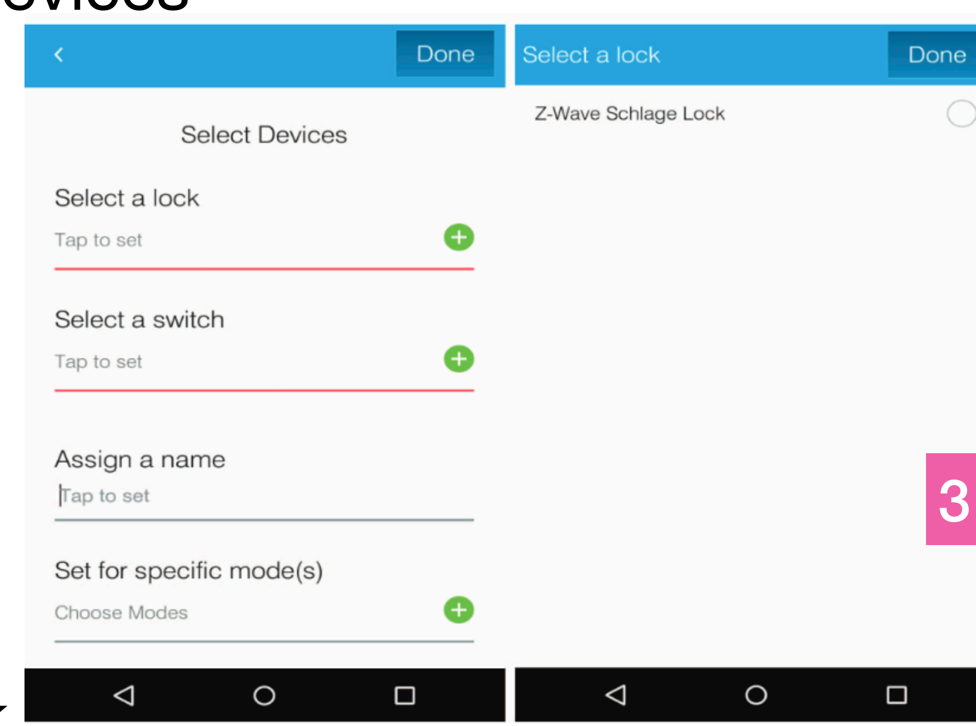
Capabilities requested in a SmartApp.

Device Handlers Software-wrappers for physical devices



1

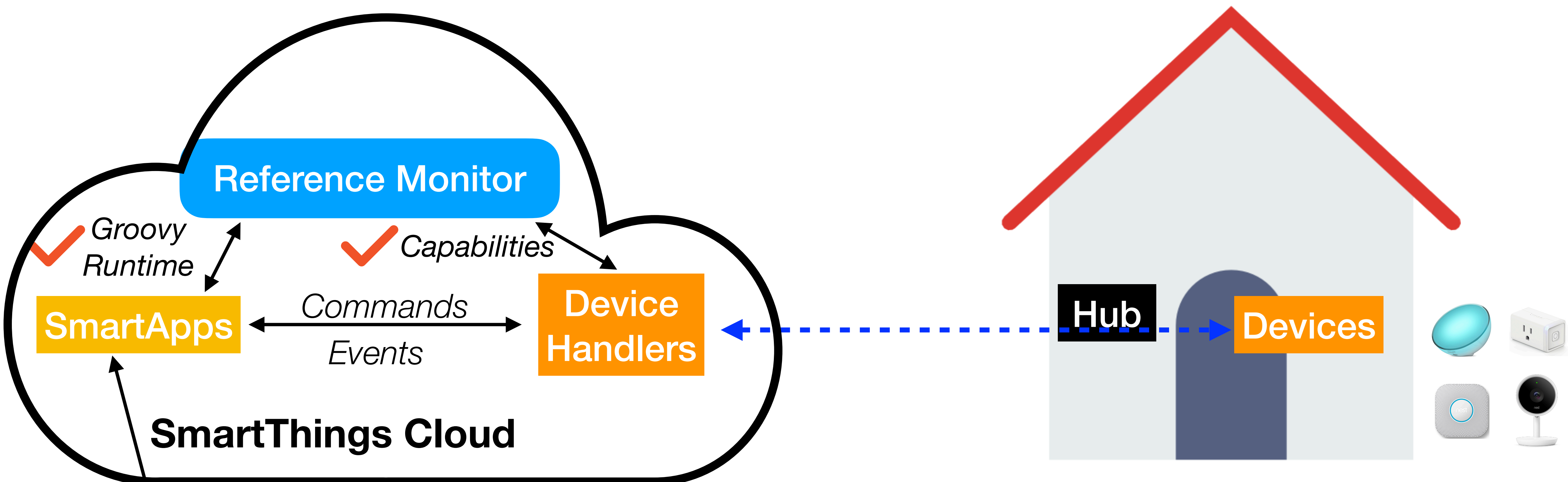
Capabilities declared in a typical door lock

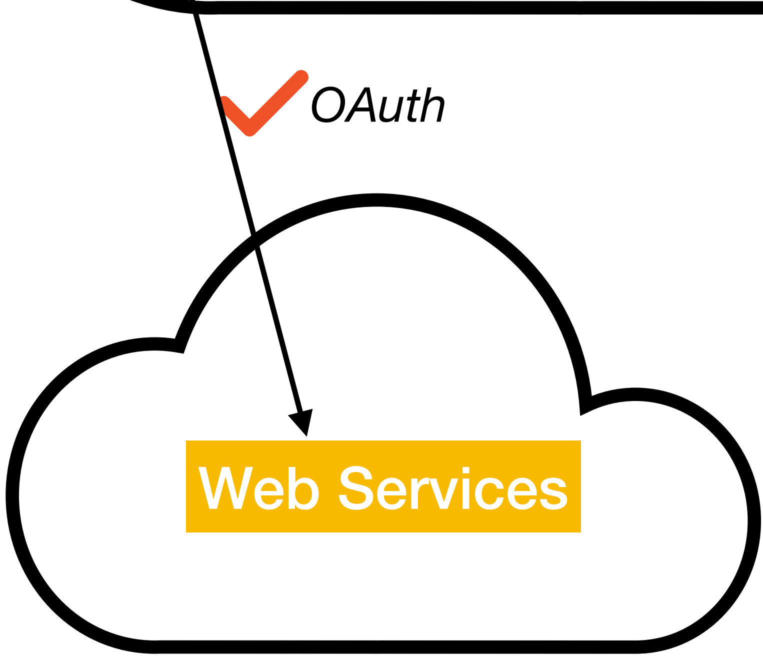


3

Background: SmartThings

- SmartThings uses both the hub and the cloud (pre-2019)





- **Eg Sec Policy:** Permission assignment to apps
- **Reference monitor:** The *mechanism* that ensures that the security *policy* is enforced correctly
- **Reference Monitor Properties**
  - Complete Mediation
  - Tamperproofness
  - Verifiability

Motivation

Key question: *Is the platform's API secure?*

Integrity

Can attackers manipulate devices? (e.g., insert lock codes)

Availability

Can attackers disable devices? (e.g., turn OFF a camera)

Privacy

Can attackers learn private information? (e.g., the user's schedule)

Authenticity

Can attackers spoof messages? (e.g., event spoofing, using stolen OAuth tokens)

Confidentiality

Can attackers learn sensitive information (e.g., lock codes)

Methodology

- Dynamic Testing
- Static Analysis
  - Source code (Groovy SmartApps)
  - Binaries (certain Android apps)
- Network Analysis (mainly to build the dataset)
- **Research Questions:**
  - How *overprivileged* are apps?
  - Can events be *spoofed*?
  - What sensitive information can apps access?
  - How do external third-party integrations affect security?
  - …

Findings

- Overprivilege
- Event injection (*i.e., spoofing)*
- Event Sniffing
- Vulnerable Third-party integrations

Findings: Overprivilege

- **Coarse-grained Capabilities** Policy
  - App asks for capability "lock"
    - Can read the lock's state, and issue the "lock" and "unlock" commands.
  - *What if the app only needs to read the lock state?*

- **Device-granularity binding** Mechanism
  - Apps get *all* capabilities for a device, if they ask for just one.

Which of these is a policy problem, vs a mechanism problem?

Which of these would be harder to fix?

Findings: Event Injection

- **Dynamic code loading**
  - SmartApps use dynamic method invocation
  - Can be exploited to execute any code in the SmartApp's *security context (i.e., the capabilities available to the SmartApp)*

```
def updateDevice() {
  def data = request.JSON
    def command = data.command
    def arguments = data.arguments

    log.debug "updateDevice, params: ${params}, request: ${data}"
    if (!command) {
      render status: 400, data: '{"msg": "command is required"}'
    } else {
      def device = allDevices.find { it.id == params.id }
      if (device) {
        if (arguments) {
          device."$command"(*arguments)
        } else {
          device."$command"()
        }
        render status: 204, data: "{}"
      } else {
        render status: 404, data: '{"msg": "Device not found"}'
      }
```

Findings: Event Injection

- **Dynamic code loading**
  - SmartApps use dynamic method invocation
  - Can be exploited to execute any code in the SmartApp's *security context (i.e., the capabilities available to the SmartApp)*
- **Event spoofing is trivially possible**
  - Direct Approach: Spoof an event message, with the 128 bit ID of the device
  - Indirect Approach: Modify the *locationMode*. No access control policy protecting it!

Findings: Sniffing

- **A SmartApp can listen to *everything* from a *bound* device**
  - No access control in place
  - Can subscribe to all events, if binding is established.

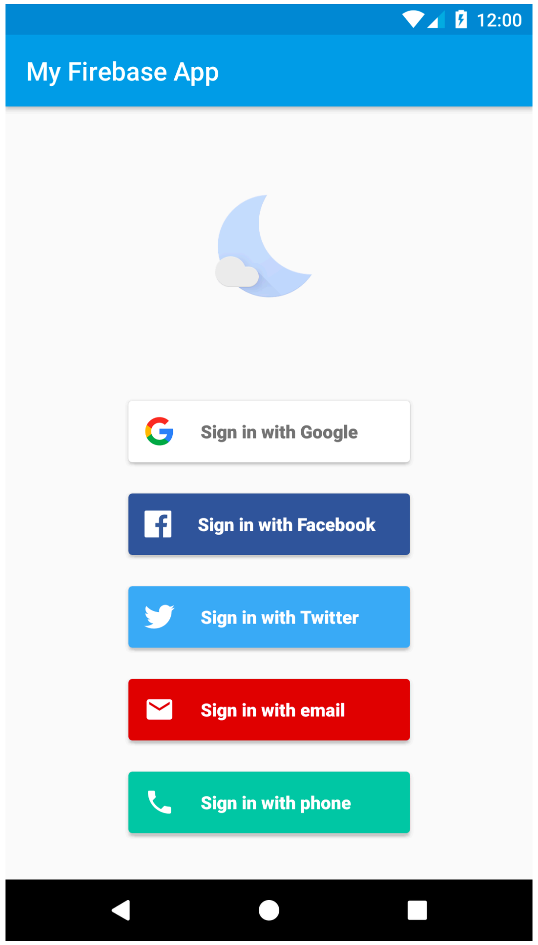
- **A SmartApp can listen to *everything* if it knows the 128 bit *device ID***
  - Even if the device is not bound to the SmartApp

Why is this bad?

How can the adversary get this device ID?

Findings: Vulnerable 3rd Party Integrations

- ***OAuth tokens* can be stolen, or rather, *falsely acquired***
  - OAuth tokens enable a 3rd-party to connect to the user's SmartThings account.
  - To successfully acquire an OAuth token for a user's SmartThings account, a Web service needs:
    1. a *client ID*
    2. a *client secret*
    3. the user to sign in, and redirect a *code* to the Web service.
  - Mobile apps often hardcode the client ID and secret, and reduce the barriers to acquiring a token.



Attack!

1. **Inject Key Codes!**
    1. Acquire (Steal) Token + Inject Commands (using capabilities not requested)
2. **Pin Code Snooping:**
    1. Acquire device ID or bind to the device (e.g., battery monitor) + register for certain events (e.g., CodeReport)
3. **Disabling Vacation Mode** (*what's the harm?*)
4. **Fake Alarm** (*what's the harm?*)

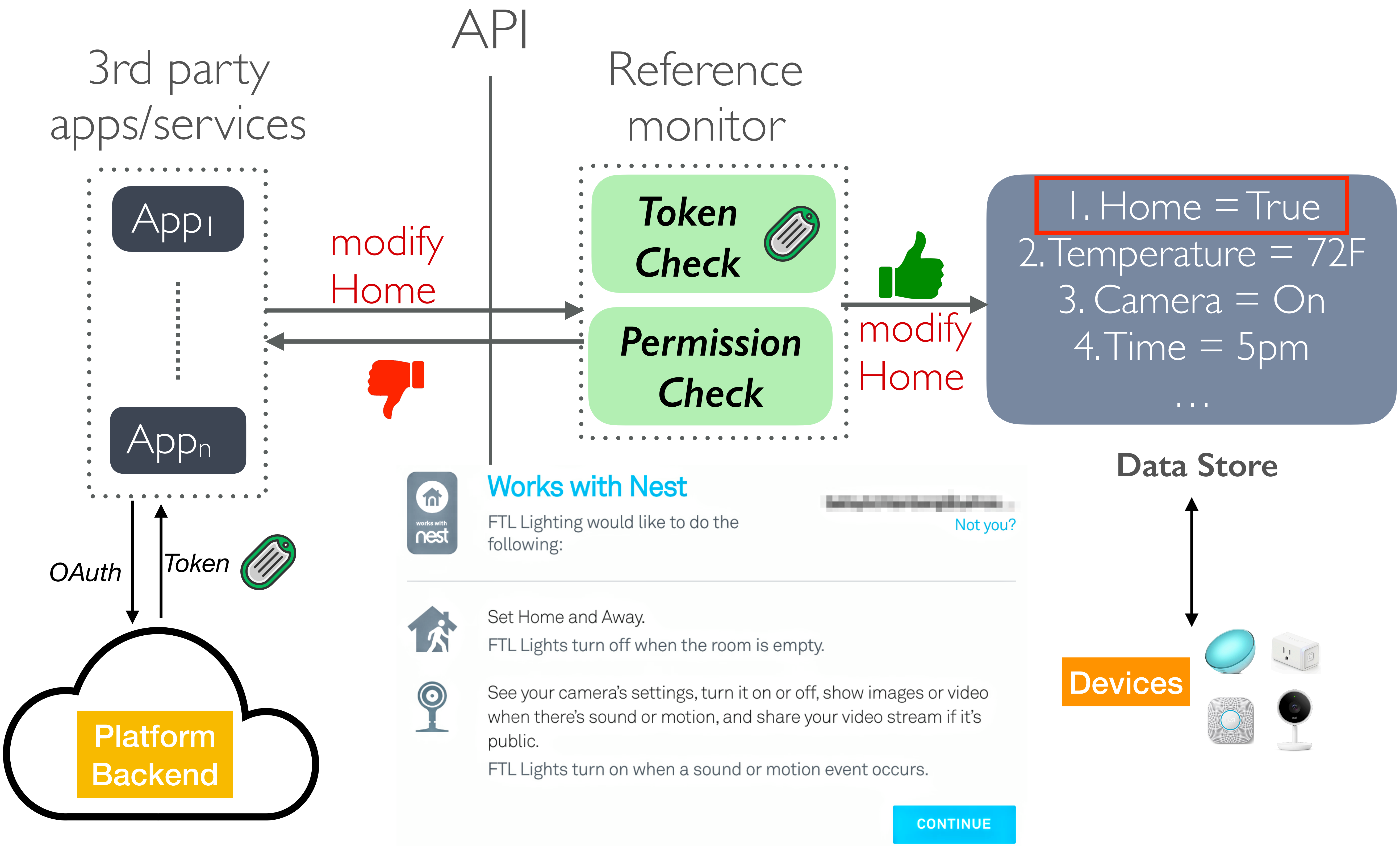
Platforms

SmartThings
(pre-2019)

Google Nest

Philips Hue

Background: Nest/Hue

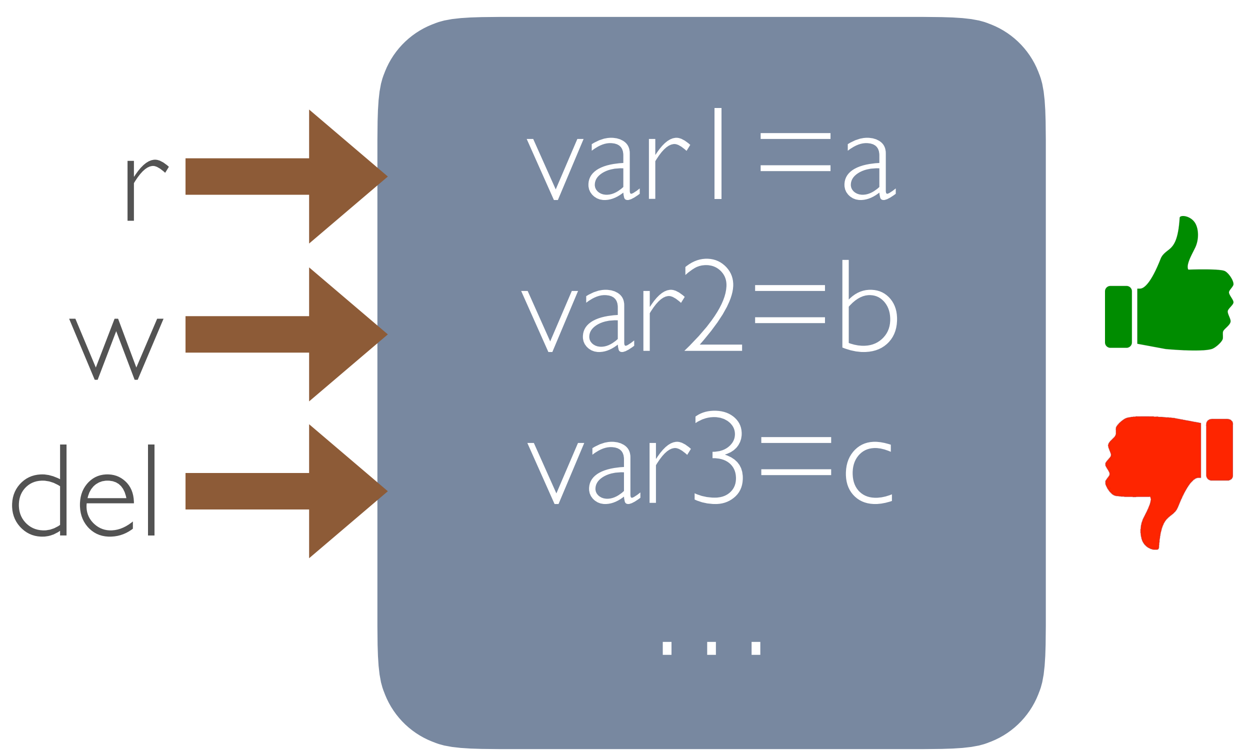


Methodology

- Permission Map generation
- Static Analysis
  - Source code (third-party apps)
- Dynamic analysis
  - SSL implementation
- **Research Questions:**
  - Access control correctly enforced i.e., bypassing permissions?
  - Apps overprivileged?
  - How do external third-party integrations affect security?
  - …

Methodology

- Are the platforms enforcing permissions correctly?
- Using automatically generated permission maps!



- Permission1 -> Var1 (r,w), var2(r)
- Permission2 -> var2 (r), var3 (r)
- ....

Findings: Permission Enforcement

Enforces permissions correctly, i.e., as described in the documentation

- Can **bypass user consent!**

linkbutton	bool	Indicates whether the link button has been pressed within the last 30 seconds. Starting `1.31`, Writing is only allowed for Portal access via cloud application_key.

Findings: Permission Enforcement

nest

Enforces permissions correctly, i.e., as described in the documentation

hue

- Can **bypass user consent!**
- Can **add/remove other apps!**

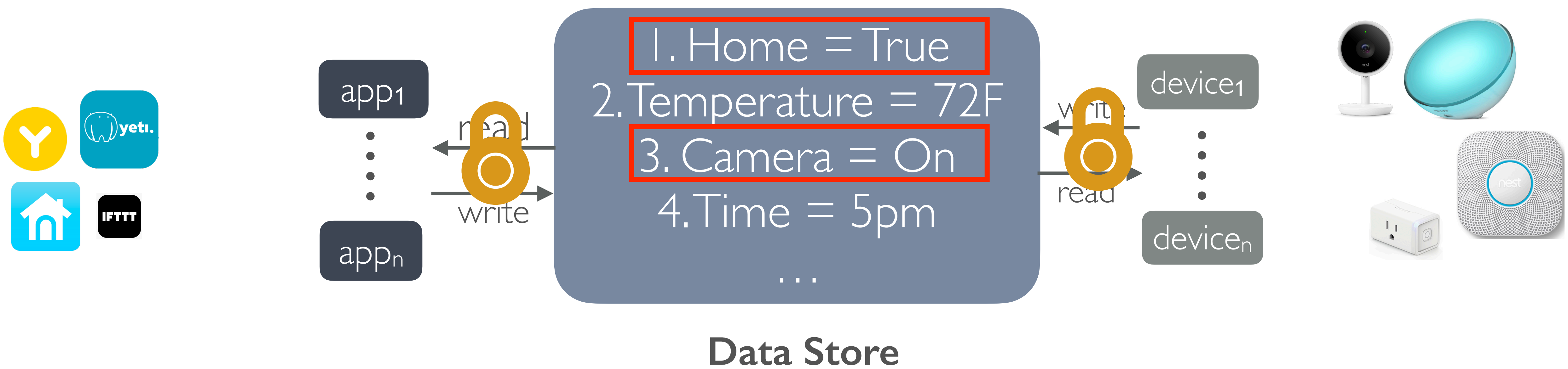
7.4. Delete user from whitelist

URL	/api/<application_key>/config/whitelist/<element>
Method	DELETE
Version	1.0
Permission	Whitelist; Starting 1.31.0: Only via https://account.meethue.com/apps

Attacks using Routines: Lateral Privilege Escalation

Recall how routines work

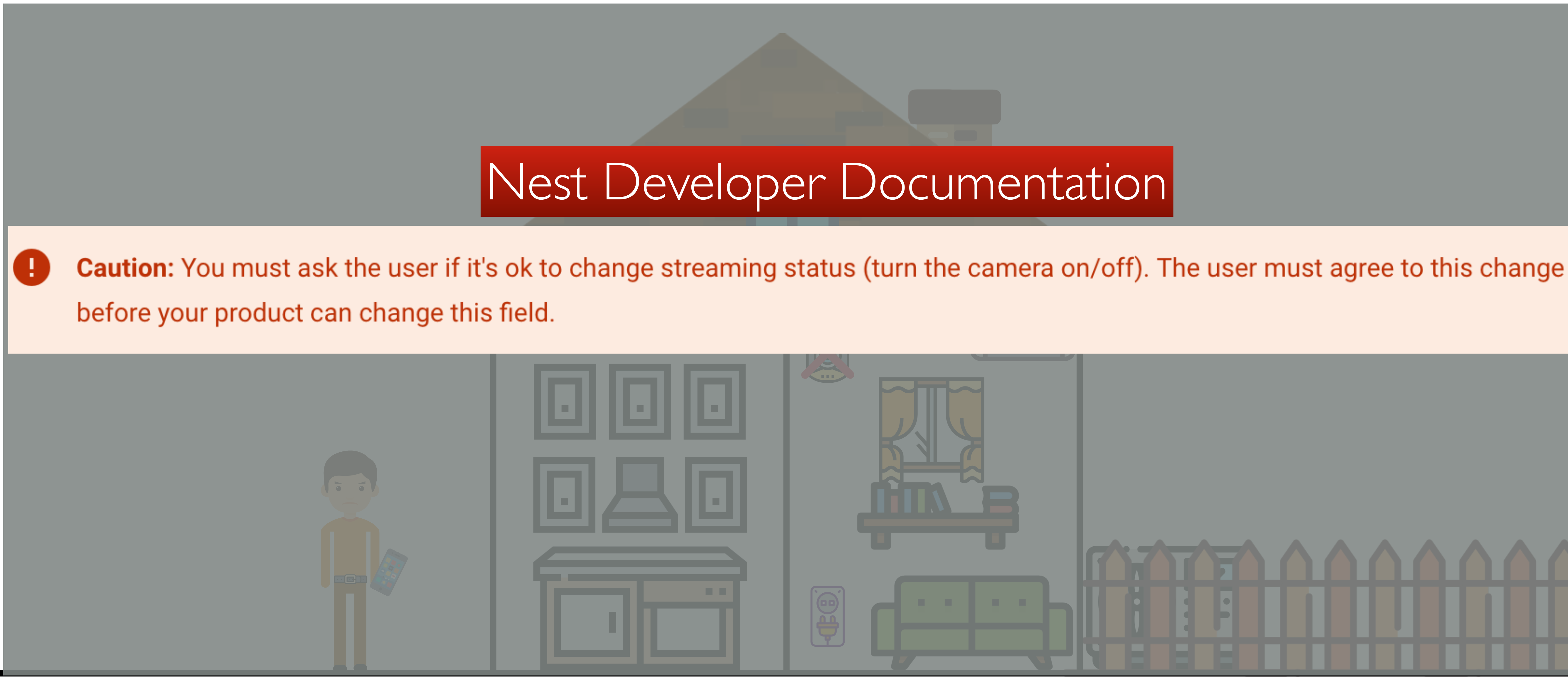
Data Store-Based (DSB) platforms



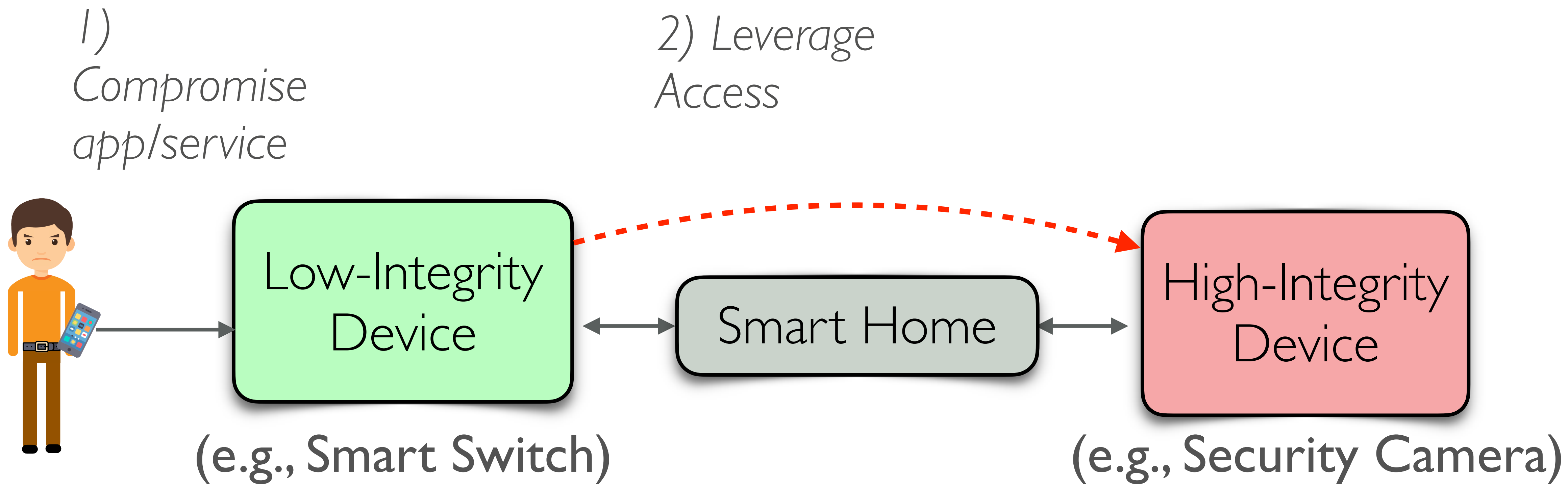
Permissions protect reads/writes to high-security variables (e.g., Camera ON/OFF, user home/away)

Hypothetical Scenario

Hypothetical Scenario



Lateral Privilege Escalation



ANALYSIS OVERVIEW

Analysis: Apps

Secure Communication?

Analysis: Routines

Vulnerable to Attacks?

ANALYSIS: APPS

Analyzed the SSL connections in apps using *Mallodroid*[1]

650 *General smart home apps*

20.61% with at least one SSL issue (134/650)

111 *'Works with Nest' apps*

19.82% with at least one SSL issue (22/111)

Most common causes:
TrustManager - 20
HostNameVerifier - 11

Accept all certificates! → TrustManager

Don't verify hostname of signed certificates! → HostNameVerifier

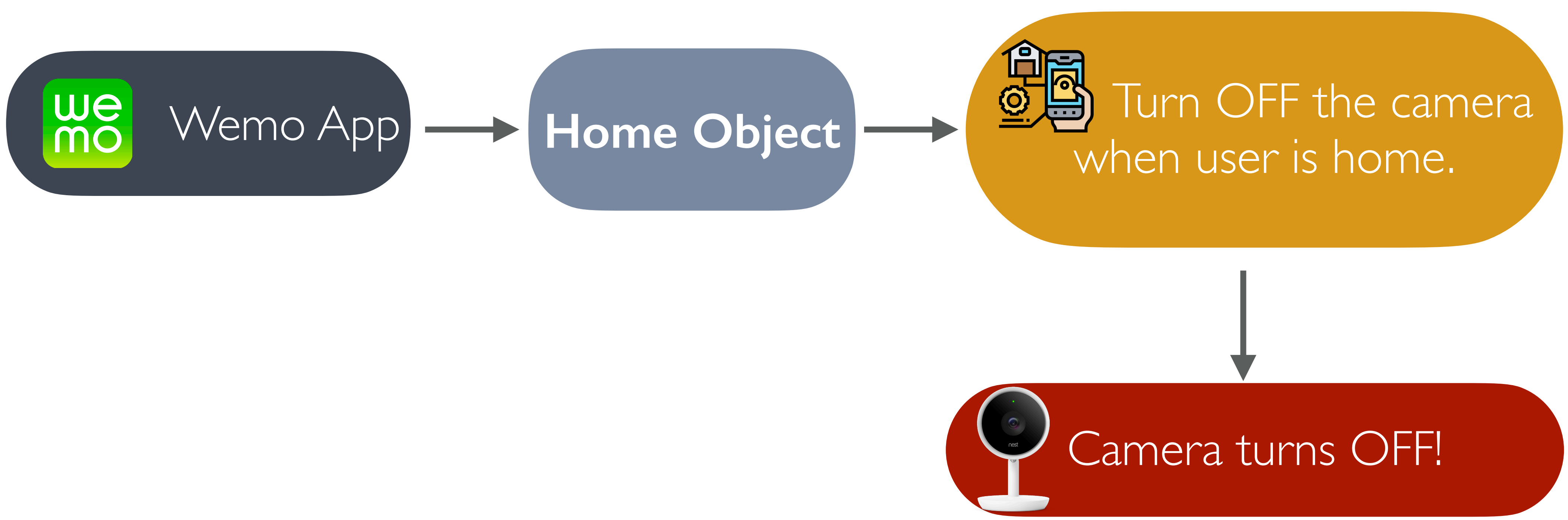
1. Fahl, Sascha, et al. "Why Eve and Mallory love Android: An analysis of Android SSL (in) security." *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

ANALYSIS: ROUTINES

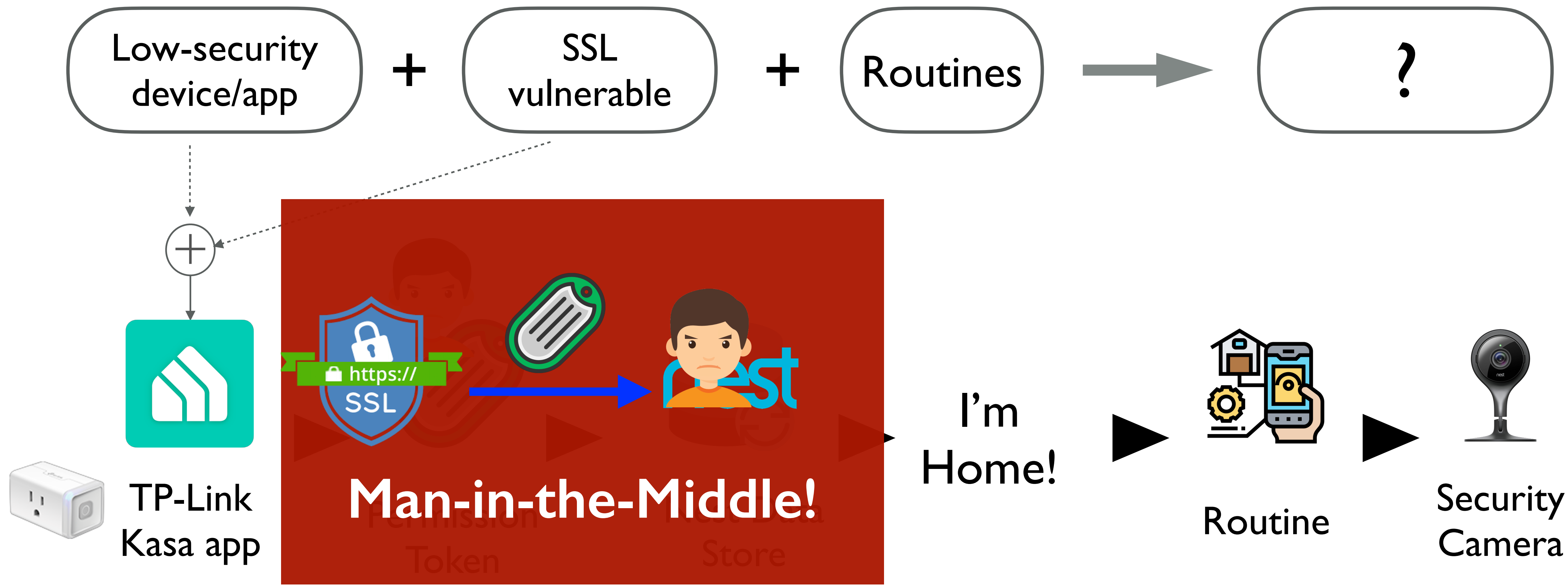
nest

Heterogeneous set of devices

Diverse and expressive routines



Putting it All Together



Successful Lateral Privilege Escalation

Suggestions/Discussion

- *Risk-based capabilities* would prevent overprivilege.
  - User-studies to quantify risk
- *App and Device Identity* to prevent event spoofing
  - Any crypto applications?
  - Similar approaches to using UID in Android?
- *A unified security perspective across platforms* (mobile apps and smart home) to identify the impact of vulnerable integrations
  - Security-critical devices may be *dependent* on other system components to be truly secure.
  - Adversaries can leverage seemingly *disconnected* components to create an attack.