# Security in Centralized Data Store-based Home Automation Platforms: A Systematic Analysis of Nest and Hue



KAUSHAL KAFLE, William & Mary, United States of America
KEVIN MORAN, William & Mary, United States of America
SUNIL MANANDHAR, William & Mary, United States of America
ADWAIT NADKARNI, William & Mary, United States of America
DENYS POSHYVANYK, William & Mary, United States of America

Home automation platforms enable consumers to conveniently automate various physical aspects of their homes. However, the security flaws in the platforms or integrated third-party products can have serious security and safety implications for the user's physical environment. This paper describes our systematic security evaluation of two popular smart home platforms, Google's Nest platform and Philips Hue, which implement home automation "routines" (*i.e.,* trigger-action programs involving apps and devices) via manipulation of state variables in a *centralized data store*. Our semi-automated analysis examines, among other things, platform access control enforcement, the rigor of non-system enforcement procedures, and the potential for misuse of routines, and leads to *eleven* key findings with serious security implications. We combine several of the vulnerabilities we find to demonstrate the first end-to-end instance of lateral privilege escalation in the smart home, wherein we remotely disable the Nest Security Camera via a compromised light switch app. Finally, we discuss potential defenses, and the impact of the continuous evolution of smart home platforms on the practicality of security analysis. Our findings draw attention to the unique security challenges of smart home platforms, and highlight the importance of enforcing security by design.

Additional Key Words and Phrases: Smart Home; Routines; Lateral Privilege Escalation; Overprivilege

**ACM Reference Format:**
Kaushal Kafle, Kevin Moran, Sunil Manandhar, Adwait Nadkarni, and Denys Poshyvanyk. 2020. Security in Centralized Data Store-based Home Automation Platforms: A Systematic Analysis of Nest and Hue. *ACM Transactions on Cyber-Physical Systems* 5, 1, Article 2 (December 2020), 25 pages. https://doi.org/10.1145/3418286

## 1 INTRODUCTION

Internet-connected, embedded computing objects known as *smart home products* have become extremely popular with consumers. The utility and practicality afforded by these devices has spurred tremendous market interest, with over 20 billion smart home products projected to be in use by 2020 [19]. The diversity of these products is staggering, ranging from small physical devices with embedded computers such as smart locks and light bulbs, to full fledged appliances such as refrigerators and HVAC systems. In the modern computing landscape, smart home devices are unique as they provide an often imperceptible bridge between the digital and physical worlds by connecting physical objects to digital services via the Internet, allowing the user to conveniently

Authors' addresses: Kaushal Kafle, kkafle@cs.wm.edu; Kevin Moran, kpmoran@cs.wm.edu, Sunil Manandhar, sunil@cs.wm.edu; Adwait Nadkarni, nadkarni@cs.wm.edu; Denys Poshyvanyk, denys@cs.wm.edu; All authors share the following affiliation: Department of Computer Science, P.O. Box 8795, Williamsburg, Virginia, United States of America.


XXXX-XXXX/2020/12-ART2 $15.00
https://doi.org/10.1145/3418286

automate their home. However, because many of these products are tied to the user's security or privacy (*e.g.,* door locks, cameras), it is important to understand the attack surface of such devices and platforms, in order to build practical defenses without sacrificing utility.

As the market for smart home devices has continued to mature, a new software paradigm has emerged to enable home automation via the interactions between smart home devices and the apps that control them. These interactions may be expressed as *routines*, which are sequences of app and device actions that are executed upon one or more triggers, *i.e.,* an instance of the trigger-action paradigm in the smart home. Routines are the building block of home automation [14, 46, 57, 58], and hence, it is natural to leverage routines to characterize existing platforms.

If we categorize available platforms based on how routines are facilitated, we observe two broad categories: (1) API-based Smart Home Managers such as Yeti [65], Yonomi [66], IFTTT [25], and Stringify [55] that allow users to chain together a diverse set of devices using APIs exposed by device vendors, and (2) platforms such as Google's Works with Nest [37], Samsung SmartThings [52], and Philips Hue [44] that leverage *centralized data stores* to monitor and maintain the states of IoT devices. We term these platforms as Data Store-Based (DSB) Smart Home Platforms. In DSB platforms, complex routines are executed via reads/writes to state variables in a central data store.

This paper is motivated by a key observation that while routines are supported via centralized data stores in all DSB platforms, there are differences in the manner in which routines are created, observed, and managed by the user. That is, SmartThings encourages users to take full control of creating and managing routines involving third-party apps and devices via the SmartThings app. On the other hand, in Nest, users do not have a centralized perspective of routines at all, and instead, manage routines using third-party apps/devices. This key difference may imply unique security challenges for Nest. Similarly, being a much simpler platform within this category of DSB platforms, Hue represents another unique and interesting instance of the DSB platform paradigm.

**Contributions:** This paper performs a systematic security analysis of some of the less studied, but widely popular, data store-based smart home platforms, *i.e.,* Nest and Hue. In particular, we evaluate (1) the access control enforcement in the platforms themselves, (2) the robustness of other non-system enforcement (*e.g.,* product reviews in Nest), (3) the use, and more importantly, the *misuse* of routines via manipulation of the data store by low-integrity devices,[1] and finally, (4) the security of applications that integrate into these platforms.

To our knowledge, this paper is the first to analyze this relatively new class of smart home platforms, in particular the Nest and Hue platforms, and to provide a holistic analysis of routines, their use, and potential for their misuse in DSB platforms. Moreover, this paper is the first to analyze the accuracy of app-defined permission descriptions and prompts, which provide highly critical context to the user. Furthermore, we provide a detailed account of our vulnerability disclosure experience with four separate vendors, and discover that certain vulnerabilities may not always be fixable. Finally, we study the *ramifications of platform evolution* on the transparency and artifacts required for security analysis. In doing so we not only discover concrete problems in DSB platforms, but also use empirical analysis to reveal challenges for feasibly performing similar research studies in the near future. Our novel findings ($\mathcal{F}_1 \rightarrow \mathcal{F}_{11}$) are summarized as follows:

- **Misuse of routines** – The permission model in Nest is fine-grained and enforced according to specifications ($\mathcal{F}_1$), giving low-integrity third-party apps/devices (*e.g.,* a switch) little room for directly modifying the data store variables of high-integrity devices (*e.g.,* security cameras). However, routines supported by Nest allow low-integrity devices/apps to indirectly modify the state of high-integrity devices, by modifying the shared variables they rely on ($\mathcal{F}_4$).

[1] In the context of our study, we define a device as high-integrity if it is advertised as security-critical by the device vendor (*e.g.,* Nest Cam) while those that are not security-critical are referred to as low-integrity (*e.g.,* Philips Hue lamp).

- **Lack of systematic defenses** – Nest does not employ transitive access control enforcement to prevent indirect modification of security-sensitive data store variables; instead, it relies on a product review of application artifacts before allowing API access. We discover that the product review process is insufficient and may not prevent malicious exploitation of routines; *i.e.,* the review mandates that apps prompt the user before modifying certain variables, but does not validate *what* the prompt contains, allowing apps to deceive users into providing consent ($\mathcal{F}_5$). Moreover, permission descriptions provided by apps during authorization are also often incorrect or misleading ($\mathcal{F}_6$, $\mathcal{F}_9$), which demonstrates that malicious apps may easily find ways to gain more privilege than necessary ($\mathcal{F}_7$), circumventing both users and the Nest product review ($\mathcal{F}_8$).
- **Lateral privilege escalation** – We find that smart home apps, particularly those that connect to Nest and have permissions to access security-sensitive data store variables, have a significantly high rate of SSL vulnerabilities ($\mathcal{F}_{10}$). We combine these SSL flaws with the findings discussed previously (specifically $\mathcal{F}_4 \rightarrow \mathcal{F}_9$) and demonstrate a novel form of a *lateral* privilege escalation attack. That is, we compromise a low-integrity app that has access to the user's Nest smart home (*e.g.,* a TP Link Kasa switch), use the compromised app to change the state of the data store to trigger a security-sensitive routine, and indirectly change the state of a high-integrity Nest device (*e.g.,* the Nest security camera). This attack can be used to deceive the Nest Cam into determining that the user is home when they are actually away, effectively disabling it.
- **Lack of bare minimum protections** – Unlike Nest, the access control enforcement of Hue is woefully inadequate. Third-party apps that have been added to a user's Hue platform may arbitrarily add other apps without user consent, despite an existing policy that the user must consent to by physically pressing a button ($\mathcal{F}_2$). Making matters worse, an app may *remove* other apps integrated with the platform by exploiting unprotected data store variables in Hue ($\mathcal{F}_3$). These vulnerabilities may allow an app with seemingly useful functionality (*i.e.,* a Trojan [28]) to install malicious add-ons without the user's knowledge, and replace the user's integrated apps with malicious substitutes. While repeating our experiments on a version of Hue updated to address these issues, we discover that Hue's mitigation is only partially successful ($\mathcal{F}_{11}$).

The rest of the paper is structured as follows: Section 2 describes the key attributes of DSB platforms. Section 3 describes our security evaluation of Nest and Hue. Section 4 explores the significance of our findings (i.e., a lateral privilege escalation attack, Section 4.1), the current status of the vulnerabilities reported to the concerned vendors (Section 4.2), and challenges for future analysis, in the form of an empirical study of the feasibility of our security analyses with 6 additional smart home platforms (Section 4.3). Section 5 describes potential defenses (or preventative steps) for the attacks and vulnerabilities discussed in this paper. Section 6 describes the related work. Section 7 concludes with lessons learned.

## 2 HOME AUTOMATION VIA CENTRALIZED DATA STORES

This section describes the general characteristics of data store-based platforms, *i.e.,* smart home platforms that use a *centralized data store* to facilitate routines. We provide the background on two such platforms, namely (1) Google's "Works with Nest" [38] platform (henceforth called "Nest") and (2) the Philips Hue lighting system [44] (henceforth called "Hue"), which serve as the targets of our security evaluation. The Android apps for both of the systems have over a million downloads on Google Play [20, 21], indicating significant adoption, and far-reaching impact of our analysis.

### 2.1 General Characteristics

Figure 1 shows the general architecture of DSB platforms, consisting of 3 main components: *apps*, *devices*, and the *centralized data store*, which generally communicate over the Internet. Additionally, a physical hub that facilitates local communication via protocols such as Zigbee or Z-wave may be present (*e.g.,* the Hue Bridge). The apps may either be Web services hosted on the cloud, or mobile

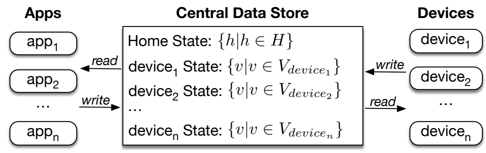


Fig. 1. The general architecture of platforms that leverage centralized data stores. Note that $H$ is the universe of all home state variables, and $V_{device_i}$ is the universe of all state variables specific to $device_i$.

apps communicating via Web services. At this juncture, we generalize apps as third-party software interacting with the data store, and provide the platform-specific descriptions later.

The centralized data store facilitates communication among apps and devices via state variables. The data store exposes two types of state variables: (1) *Home* state variables that reflect the general state of the entire smart home (*e.g.,* if the user is at *home/away*, the *devices attached* to the home, the *postal code*), and (2) *Device-specific* state variables that reflect the attributes specific to particular devices (*e.g.,* if the Camera is *streaming*, the *target temperature* of the thermostat).

Apps and devices communicate by reading from or writing to the state variables in the data store. This model allows expressive communication, from simple state updates to indirect trigger-action routines. For instance, the thermostat may change to its "economy" mode when the home's state changes to *away*, *i.e.,* the thermostat's app may detect that the user has left the smart home (*e.g.,* using Geofencing), and *write* to the home state variable *away*. The thermostat may then *read* this change, and switch to its economy mode. Our preliminary investigation led to the following *key observations* that motivate a targeted analysis of the Nest and Hue DSB platforms:

**Key observations motivating the analysis of Nest and Hue:** We observe that both Nest and SmartThings execute routines; however, there is a key difference in how routines are managed. SmartThings allows users to create and manage routines from the SmartThings app itself, thereby providing users with a general view of all the routines executing in the home [53]. In contrast, Nest routines are generally implemented as *decentralized* third-party integrations. Third-party products that facilitate routines provide the user with the ability to view and manage them. As a result, the Nest platform does not provide the user with a *centralized view* of the routines that are in place. Due to this lack of user control, Nest smart homes may face unique security risks and challenges, which motivates its security analysis. Similarly, we analyze Hue as it is an interesting variant of DSB platforms, and a relatively popular brand [21]. That is, Hue integrates *homogeneous* devices related to lighting such as lamps and bulbs, unlike Nest and SmartThings that integrate heterogeneous devices, and represents a drastically simpler (and hence unique) variant of home automation platforms that use centralized data stores. Thus, the analysis of Hue's attack surface has potential to draw attention to other similar, homogeneous platforms, which is especially important considering the fragmentation in the smart home product ecosystem [11].

## 2.2 Nest Background

The *Works with Nest* platform integrates a heterogenous set of devices, including devices from Nest (*e.g.,* Nest thermostat, Nest Cam, Nest Protect) as well as from other brands (*e.g.,* Wemo and Kasa switches, Google Home, MyQ Chamberlain garage door opener) [38]. This section describes the key characteristics of Nest, *i.e.,* its data store, its access control model, and routines.

**Data store composition:** Figure 2 shows a simplified, conceptual view of the centralized data store in Nest. Note that the figure shows a small fraction of the true data store, *i.e.,* only enough to facilitate understanding. Nest implements the data store as a JSON-format document divided into two main top-level sections: *structures* and *devices.* A *structure* represents an entire smart home environment such as a user's home or office, and is defined by various state variables that are global

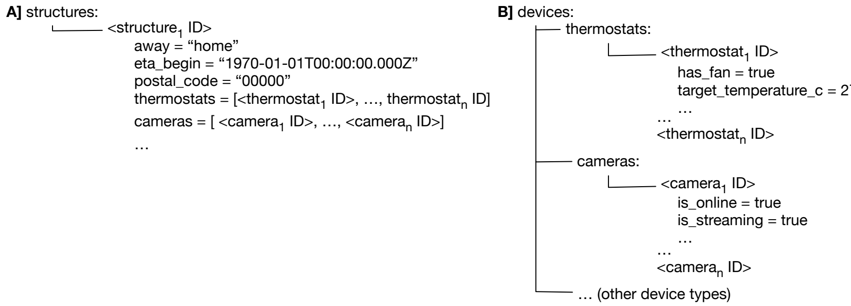

Fig. 2. A simplified view of the centralized data store in Nest.

across the smart home (*e.g., Away* to indicate the presence or absence of the user in the structure and the *postal_code* to indicate the home's physical location). The devices are subdivided into device types (*e.g.,* thermostats, cameras, smoke detectors), and there can be many devices of a certain type, as shown in Figure 2. Each device stores its state in variables that are relevant to its type; *e.g.,* a thermostat has state variables for *humidity*, and *target_temperature_c*, whereas a camera has the variables *is_online* and *is_streaming*. Aside from these type-specific variables, devices also have certain variables in common; *e.g.,* the alphanumeric *device ID*, the *structure ID* of the structure in which the device is installed, the device's user-assigned *name*, and *battery_health*.

**Access Control in Nest:** Nest treats third-party apps, Web services, and devices that want to integrate with a Nest-based smart home as "products". Nest defines read or read/write permissions for each of the variables in the data store. A product that wants to register with Nest must first declare the permissions that it needs (*e.g., thermostat read/write*) in the Nest developer console. Each Nest user account has a specific data store assigned to it, and products gain access to the data store using OAuth 2.0. That is, users connect products to their account by (1) authenticating with Nest, and (2) allowing the third-party app/product to use certain Nest permissions that it requests using an install-time permission prompt. Once the user grants the permissions, a revocable access token is generated that is specific to the product, the set of permissions requested, and the particular smart home. This token is required for subsequent interactions with the data store, and hence, must be protected from attackers. The attack described in this paper (Section 4.1) relies on the attacker's ability to steal the token; however, we do not tamper with the OAuth authorization itself, and instead steal the token via other channels (*e.g.,* MiTM attack exploiting SSL misuse).

**Accessing the Nest data store:** Devices and applications that are connected to a particular smart home (*i.e.,* the user's Nest account) can update data store variables to which they have access, and also subscribe to the changes to the state of the data store. Nest uses the REST approach for these update communications, as well as for apps/devices to modify the data store. The REST endpoints can be accessed through HTTPS by any registered Nest products.

**Routines in Nest:** In Nest, the user cannot create or view routines in a centralized interface (*i.e.,* unlike SmartThings). Instead, apps may provide routines as opt-in features. For example, the Nest smoke alarm's *smoke_alarm_state* variable has three possible values, "ok", "warning", and "emergency". When this variable is changed to "warning", other smart home products (e.g., Somfy Protect [64]) can be configured to trigger and warn the user.

### 2.3 Hue Background

Hue implements its data store as a JSON document with sections related to (1) physical lighting devices, (2) semantic groups of these devices, and (3) global config variables (such as *whitelisted apps* and the *linkbutton*). To connect a third-party management app to a user's existing Hue system, the

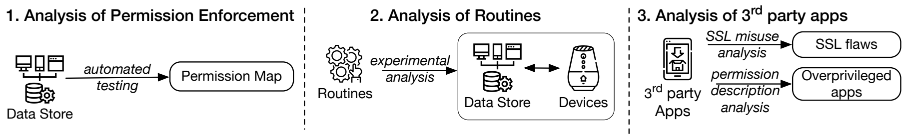


Fig. 3. An overview of the three components of our security evaluation of Nest and Hue.

app identifies a Hue bridge connected to the local network, and requires the user to press a physical button on the bridge. Once this action is completed by the user, the app receives a *username* token that is stored in the *whitelisted* section of the Hue data store. Whitelisted apps can read/modify data store variables as per Hue's access control policy, which grants all authorized apps the same access. Our online appendix provides additional details [1].

## 3 SECURITY EVALUATION OF PLATFORM PERMISSIONS, ROUTINES, AND APPS

As described in Section 2, DSB platforms consist of (1) *third-party apps* that interact with the smart home (*i.e.,* centralized data store and devices) by acquiring (2) *platform permissions*, and execute a complex set of such interactions as (3) *trigger-action routines*. Our analysis methodology takes these three aspects into consideration, as show in Figure 3, and summarized as follows:

**A. Analysis of Platform Permissions (Section 3.1):** We analyze the *enforcement* of platform permissions/access control to discover inconsistencies by automatically building permission maps.

**B. Analysis of Routines (Section 3.2):** While analyzing permission enforcement shows us what individual devices can accomplish with a certain set of permissions, we perform an experimental analysis with real devices to identify the interdependencies among devices and apps through the shared data model, and the ramifications of such interdependencies on the user's security. Additionally, Nest does not enforce transitive access control to prevent dangerous side-effects of routines, but instead employs a product review process as a defense mechanism. We analyze the effectiveness of this review process using the permission prompts used by existing apps as evidence.

**C. Analysis of Third-party Apps (Section 3.3):** We analyze the permission descriptions presented by mobile apps compatible with Nest to identify over-privileged apps, or apps whose permission descriptions are inconsistent with the permission requested. We then analyze the apps for signs of SSL misuse, which we will further leverage to indirectly exploit security critical devices.

We combine the findings from these three analyses to demonstrate an instance of a *lateral privilege escalation* attack in a smart home (Section 4.1).

### 3.1 Evaluating Permission Enforcement

The centralized data store described in Section 2 may contain variables whose secrecy or integrity is crucial; *e.g.,* unprotected write access to the *web_url* field of the camera may allow a malicious app to launch a phishing attack, by replacing the URL in the field with an attacker-controlled one. To understand if appropriate barriers are in place to protect such sensitive variables, we perform an analysis of the permission enforcement in Nest and Hue.

Our approach is to generate and analyze the *permission map* for each platform, *i.e.,* the variables that can be accessed with each permission, and inversely, the permissions needed to access each variable of the data store. Note that while this information should ideally be available in the platform documentation, prior analysis of similar systems has demonstrated that the documentation may not always be complete or correct in this regard [16, 18].

*3.1.1 Generating Permission Maps.* We generate the permission map using automated testing as in prior work on Android [16]. We use two separate approaches for Nest and Hue, owing to their disparate access control models.

**Approach for Nest:** We first created a simulated home environment using the Nest Home Simulator,[2] and linked our Nest user account to it. We then created our test Android app, and connected it to our Nest user account using OAuth, as described in Section 2.2. Note that the simulated smart home is virtually identical to an end-user's setup, and using the simulator allows us to investigate the permission enforcement for Nest devices (*e.g.,* the Smoke/CO detector).

In order to generate a complete view of the data store, we granted our test app all of the 15 permissions in Nest, and read all accompanying information. To build the permission map for Nest's 15 permissions, we created 15 apps, such that each app requested a single unique permission, and registered these apps to our developer account in the Nest developer console. Note that we do not test the effect of permission combinations, as our goal is to test the enforcement of individual permissions, and Nest's simple authorization logic simply provides an app with a union of the privileges of the individual permissions.

We then connected each of the 15 apps to our Nest user account using the procedure described in Section 2.2. We programmed each app to attempt to read and write each variable of the data store (*i.e.,* the previously derived *complete view*). We recorded the outcome of each access, *i.e.,* if it was successful, or an access control denial. In the cases where we experienced non-security errors writing to data store variables (*e.g.,* writing data with an incorrect type), we revised our apps and repeated the test. The outcome of this process was a permission map, *i.e.,* the mapping of each permission to the data store variables that it can read and/or write.

**Approach for Hue:** We followed the procedure for Hue described in Section 2.3 to get a unique token that registers our single test app with the data store of our Hue bridge. In Hue, all the variables of the data store are "readable" (*i.e.,* we verified that all the variables described in the developer documentation [3] can be read by third-party apps). Therefore, to build the permission map, we first extracted the contents of the entire data store. Then, for each subsection within the data store, our app made repeated write requests, *i.e.,* PUT calls with the payload consisting of a dummy value based on the variable type (*i.e.,* String, Boolean and Integer). All the variables that were successfully written to using this method were assigned as "writable" variables. Similarly, our app made repeated DELETE calls to the API and the variables that were successfully deleted were assigned as "writable" variables. This generated permission map applies to all third-party apps connected to Hue, since the platform provides equal privilege to all third-party apps.

*3.1.2 Analyzing Permission Maps.* The objective behind obtaining the permission map is to understand the potential for application overprivilege, by analyzing the granularity as well as the correctness of the enforcement. We analyze the permission map to identify instances of (1) *coarse-grained permissions*, *i.e.,* permissions that give the third-party app access to a set of security-sensitive resources that must ideally be protected under separate permissions, and (2) *incorrect enforcement*, *i.e.,* when an app has access to more resources (*i.e.,* state variables) than it should have given its permission set, as per the documentation; *e.g.,* apps on SmartThings may lock/unlock the door lock without the explicit permission required to do so [18].

To perform this analysis, we first identified data store variables that may be security or privacy-sensitive. This identification was performed using an open-coding methodology by one author, and separately verified by another author, for each platform. We then performed further analysis by separately considering each such variable, and the permission(s) that allow access to it. A major consideration in our analysis is the security impact of an adversary being allowed read or read/write access to a particular resource. Moreover, our evaluation of the impact of the access control enforcement was contextualized to the platform under inspection. That is, when evaluating

[2]https://developers.nest.com/documentation/cloud/home-simulator
[3]https://developers.meethue.com/philips-hue-api

Nest, we took into consideration the semantic meaning and purpose of certain permissions in terms of the data store variables, as described in the documentation (*e.g.,* that the *Away read/write* permission should be required to write to the *away* variable [33]). For Hue, we only considered the security-impact of an adversary accessing data store variables. Our rationale is that the Hue platform defines the same static policy (*i.e.,* same permissions) for all third-party apps, and hence, its permission map can be simply said to consist of just one permission that provides access to a fixed set of data store variables. As a result, we judge application over-privilege in Hue by considering the impact of an adversarial third-party app reading from or writing to each of the security-sensitive variables identified in Hue's permission map.

*3.1.3 Permission Enforcement Findings ($\mathcal{F}_1 \rightarrow \mathcal{F}_3$).* Our analysis of platform permission enforcement led to the following 3 findings:

**Finding 1: The permission enforcement in Nest is fine-grained and correctly enforced, *i.e.,* as per the specification ($\mathcal{F}_1$).** We observe that the Nest permission map is significantly more fine-grained, and permissions are correctly enforced, relative to the observations of prior research in similar platforms (*e.g.,* the analysis of SmartThings [18]). Some highly sensitive variables are always read-only (*e.g.,* the *web_url* where the camera feed is posted), and there are separate read and read/write permissions to access sensitive variables. Variables that control the state of the entire smart home are protected by dedicated permissions that control write privilege; *e.g.,* the *away* variable can only be written to using the *Away read/write* permission, the *ETA* variable has separate permissions for apps to read and write to it (*i.e., ETA read* and *ETA write*), and the Nest Cam can only be turned on/off via the *is_streaming* variable, using the *Camera + Images read/write* permission that controls write access to it. Moreover, since many apps need to respond to the *away* variable (*i.e.,* react when the user is home/away), device-specific read permissions (*e.g., Thermostat read*, *Smoke + CO read*) also allow apps to *read* the *away* variable, eliminating the need for apps to ask for higher-privileged *Away read* permission. The separate read and read/write permissions are correctly enforced, *i.e.,our generated permission map provides the same access as is defined in the Nest permission documentation [33].* This is in contrast with findings of similar analyses of permission models in the past (*e.g.,* the Android permission model [16], SmartThings [18]), and demonstrates that the Nest platform has incorporated lessons from prior work in permission enforcement.

**Finding 2: In Hue, the access control policy allows apps to bypass the user's explicit consent ($\mathcal{F}_2$).** We discovered two data store variables that were not write-protected, and which have a significant part to play in controlling access to the data store and the user's smart home. First, any third-party app can write to the *linkbutton* flag. Recall from Section 2.3 that the user has to press the physical button on the Hue bridge device to authorize an app's addition to the bridge. The physical button press changes the *linkbutton* value to "true", and allows the app to be added to the *whitelist* of allowed third-party apps. However, we discovered that once installed, an app can toggle the *linkbutton* variable at will, *enabling third-party apps to add other third-party apps to the smart home without the user's consent.* This exploitable access control vulnerability can allow an app with seemingly useful functionality to install malicious add-ons by bypassing the user altogether. In our tests, we verified this attack with apps that were connected to the local network. This condition is feasible as a malicious app that needs to be added without the user's consent may not even have to pretend to work with Hue; all it needs is to be connected to the local network (*i.e.,* a game on the mobile device from one of the people present in the smart home). Note that it is also possible to remotely perform this attack, as we discuss in Section 4.2 ($\mathcal{F}_{11}$).

**Finding 3. In Hue, third-party apps can directly modify the list of added apps, adding and revoking access without user consent ($\mathcal{F}_3$).** Hue stores the authorization tokens of apps connected to the particular smart home in a *whitelist* on the Hue Bridge device. While analyzing the permission map, we discovered that not only could our third-party test app read from this list, it could also directly delete tokens from it. We experimentally confirmed this finding again, by removing *Alexa* and *Google Home* from the smart home, without the user's consent. An adversary could easily combine this vulnerability with ($\mathcal{F}_2$), to remove legitimate apps added by the user, add adversary-controlled apps (*i.e.,* by keeping the *linkbutton* "true"), all without the user's consent. More importantly, users do not get alerts when such changes are made (*i.e.,* since it is assumed that the enforcement will correctly acquire user consent). Hence, unless the user actually checks the list of integrated apps using the Hue Web app, the user would not notice these changes.

While the Nest permission model is robust in its mapping of data store variables and permissions required to access them, Section 3.2 demonstrates how fields disallowed by permissions may be indirectly modified via strategic misuse of routines, and describes Nest's product review guidelines to prevent the same [35]. Section 3.3 describes how badly written and overprivileged apps escape these review guidelines, and motivate a technical solution.

## 3.2 Evaluating Smart Home Routines

Prior work has demonstrated that in platforms that favor application interoperability but lack transitive access control enforcement, problems such as confused deputy and application collusion may persist [9, 17, 30, 31]. Smart homes that facilitate routines are no different, but the exploitability and impact of routines on smart homes is unknown, which motivates this aspect of our study.

Recall that routines are trigger-action programs that are either triggered by a change in some variable of the data store, or whose action modifies certain variables of the data store. While both Nest and Hue share this characteristic, routines in Hue are fairly limited in scope, and their exploitation is bound to only affect the lighting of the smart home. As a result, the security analysis in this section is focused on the heterogeneous Nest platform that facilitates more diverse routines.

*3.2.1 Methodology for the Analysis of Routines.* While using the simulator as described in Section 3.1 allows us to understand what routines are *possible* on the platform, *i.e.,* what variables might be manipulated, and what Nest devices are affected as a result, we performed additional experiments with real apps and devices to study existing routines in the wild. For this experiment, we extended the smart home setup previously discussed in Section 3.1 with real devices.

We started by collecting a list of devices that integrate with Nest from the *Works with Nest* website [38]. Using this initial list and information from the website, we purchased a set of 7 devices that possessed a set of characteristics relevant to this study, *i.e.,* devices that (1) take part in routines (*i.e.,* as advertised on the website), (2) are important for the user's security or privacy, and (3) are widely-known/popular with a large user base (*i.e.,* determined by the number of installs of the mobile client on Google Play). We obtained a final list of devices (7 real and 2 simulated) connected to our Nest smart home, namely, the Nest Cam, Hue light bulb, Belkin Wemo switch, MyQ Chamberlain garage door opener, TP Link Kasa Smart Plug, Google Home, Alexa, Nest Thermostat (simulated), and the Nest Protect Smoke & CO Alarm (simulated).

We connected these devices to our Nest smart home using the Android apps provided by device vendors, and connected a small set of smart home managers (*e.g.,* Yeti [65] and Yonomi [66]) to our Nest smart home as well. For each device, we set up and executed every routine described on the Works with Nest as well as on the device vendor's website, and observed the effects on the rest of the smart home (especially, security-sensitive devices). Also, we manipulated data store variables from our test app, and observed the effects on previously configured routines and devices.

*3.2.2 Smart Home Routine Findings ($\mathcal{F}_4 \rightarrow \mathcal{F}_5$).* Our analysis of smart home routines led to the following 2 findings:

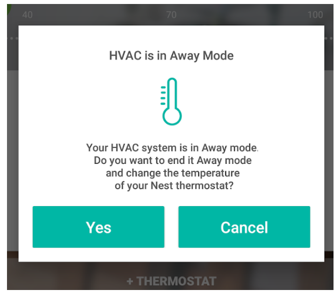


Fig. 4. The *Keen Home* app asks the user to modify the thermostat's mode, but in reality, this action leads to the *entire* smart home being set to "home" mode, which affects a number of other devices.

**Finding 4. Third-party apps that do not have the permission to turn on/off the Nest Cam directly, can do so by modifying the *away* variable ($\mathcal{F}_4$).** The Nest Cam is a home monitoring device, and important for the users' security. The *is_streaming* variable of the Nest Cam controls whether the camera is on (*i.e.,* streaming) or off, and can only be written to by an app with the permission *Camera r/w*. The Nest Cam provides a routine as a feature, which allows the camera to be automatically switched on when the user leaves the home (*i.e.,* when the *away* variable of the smart home is set to "away"), and switched off when the user returns (*i.e.,* when *away* is set to "home"). Leveraging this routine, third-party apps such as the Belkin Wemo switch can manipulate the *away* field, and indirectly affect the Nest Cam, without having explicit permission to do so. We tested this ability with our test app (see Section 3.1) as well, which could indirectly switch the camera *on* and *off* at will. This problem has serious consequences; *e.g.,* a malicious test app with the *away r/w* permission may set the variable to "home" when the user is away to prevent the camera from recording a burglary. The key problem here is that a *low-integrity device/app can trigger a change in a high-integrity device indirectly*, *i.e.,* by modifying a variable it relies on, which is an instance of the well-known information flow *integrity* problem. Moreover, this is not the only instance of a high-integrity routine that relies on *away*; *e.g.,* the Nest x Yale Lock can lock automatically when the home changes to *away* mode [63].

Nest has a basic defense to prevent such issues: application design policies that apply to apps with more than 50 users [35]. App developers are required to submit their app for a product review to the Nest team once the app reaches 50 users, and a violation of the rather strict and detailed review guidelines can result in the app being rejected from using the Nest API. One of the review policies (*i.e.,* specifically policy 5.8) states that *"Products that modify Home/Away state automatically without user confirmation or direct user action will be rejected." [35].* Nest users may be vulnerable in spite of this defense, for two reasons. First, as attacking a smart home is an attack on a user's personal space, it is feasible to assume that most attacks that exploit routines will be targeted (*e.g.,* to perform burglaries). Assuming that the adversary can use social engineering to get the user to connect a malicious app to their Nest setup, *a targeted attack on a specific user will succeed in spite of the policy*, as the app would be developed solely for the targeted user and hence will have <50 users, and be exempt from the Nest product review. Second, it is unclear how apps are checked against this policy; our next finding demonstrates a significant omission in Nest's review.

**Finding 5. Nest's product review policies dictate that the apps must prompt users before modifying *away*, but there is no official constraint on *what* the prompt may display ($\mathcal{F}_5$).** Consider an example in Figure 4, which shows one such prompt by the *Keen Home* app [62] *when the user tries to change the temperature of the thermostat.* That is, when the user tries to change the temperature of the thermostat while the *away* variable is set to "away", the app requires us to change it to "home" before the thermostat temperature can be changed. This condition is entirely unnecessary to change the temperature. More importantly, it presents the prompt to the user in a way that states that the home/away modes are specific to the HVAC alone. This is in contrast to the actual functionality of these modes, in which a change to the *away* variable affects the *entire* smart home; *i.e.,* we confirmed that the Nest Cam gets turned off as well once we agree to the prompt. It is important to note that the *Keen Home* app has successfully passed the Nest product review, and has over 1000 downloads on Google Play.[4] This case demonstrates that the Nest product review does not consider the contents of the prompt, and a malicious app may easily misinform the user and make them trigger the *away* variable to the app's advantage. Finally, in Section 3.3.1 we demonstrate that the problem of misinforming the user is not just limited to *runtime prompts* described here, but extends to application-defined *install-time permission descriptions* ($\mathcal{F}_6 \rightarrow \mathcal{F}_9$).

### 3.3 Security Analysis of Nest Apps

In this Section, we investigate the third-party apps integrated with Nest. Unlike prior work [18], we not only report the permissions requested by apps, but also analyze the permission descriptions displayed to the user at install-time. Additionally, we analyze the rate of SSL misuse by both general smart home management apps as well as apps integrated with Nest. For this section, we do not consider the Hue platform as it has a limited ecosystem of apps as compared to Nest.

We derived two datasets for this analysis, the $\text{Apps}_{general}$ dataset, which contains 650 smart home management apps extracted from Google Play, and the $\text{Apps}_{nest}$ dataset, which includes 39 apps that integrate into the Nest platform (out of the total 130 Works with Nest apps, *i.e.,* 30%). Thus, while we cannot say that our analysis and findings ($\mathcal{F}_6 \rightarrow \mathcal{F}_9$) generalize to all the apps compatible with Nest, they certainly apply to a significant minority (*i.e.,* 30%). Our online appendix [1] details our dataset collection methodology.

*3.3.1 Application Permission Descriptions.* In the Nest platform, developers provide permission descriptions that explain how an app uses a permission while registering their apps in the Nest developer console. These developer-provided descriptions are the *only* direct source of information available to the user to understand why an app requires a particular permission, *i.e.,* Nest itself only provides a short and generic permission "title" phrase that is displayed to the user along with the developer-defined description (*e.g.,* for *Thermostat read*, the Nest phrase is "See the temperature and settings on your thermostat(s)"). Owing to their significant role in the user's understanding of the permission requirements, we analyze the *correctness* of such developer-defined descriptions relative to the permissions requested.

*3.3.2 Analysis Methodology.* As described in Section 2, upon registering permissions at the developer console, developers are granted an OAuth URL that they can direct the user to for obtaining an access token. As a result, permissions are not encoded in the client mobile app or Web app (*i.e.,* unlike Android), which makes the task of extracting permissions difficult. However, we observe that the permissions that an app asks for are *always* displayed to the user for approval (*i.e.,* when first connecting an app to their Nest smart home using OAuth). We leverage this observation to obtain permissions dynamically, *i.e.,* by executing apps to the point of integrating them with our Nest smart home, and recording the permission prompt displayed for the user's approval.

*3.3.3 Nest App Findings ($\mathcal{F}_6 \rightarrow \mathcal{F}_9$).* The two permissions that dominate the permission count are *Away read/write* and *Thermostat read/write*, requested by 20 and 24 apps respectively, from the $\text{Apps}_{nest}$ dataset. Our specific findings from this analysis are as follows:

[4]https://play.google.com/store/apps/details?id=com.hipo.keen//

Table 1. Permission description violations discovered in Works with Nest apps

| Application | Incorrect Permission Description |
|---|---|
| **VC1: Requesting Read/Write instead of Read** | |
| 1. Home alerts | "**thermostat read/write:** Allows Home alerts to notify you when the Nest temperature exceeds your threshold(s)" |
| 2. Home alerts | "**away read/write:** Allows Home Alerts to notify you when someone is in your home while in away-mode" |
| 3. MyQ Chamberlain | "**thermostat read/write:** Allows Chamberlain to display your Nest Thermostat temperature in the MyQ app" |
| 4. leakSMART | "**thermostat read/write:** Allows leakSMART to show Nest Thermostat room temperature and humidity. New HVAC sensor mode will notify you to shut off your thermostat if a leak is detected in your HVAC system." |
| 5. Simplehuman Mirror | "**Camera+Images read/write:** Allow your simplehuman sensor mirror pro to capture and recreate the light your Nest Cam sees" |
| 6. Iris by Lowe's | "**structure read/write:** View your Nest Structure names so Iris can help you pair your Nest Structures to the correct Iris Places" |
| 7. Heatworks model 1 | "**away read/write:** Allows the Heatworks MODEL 1 to be placed into vacation mode to save on power consumption while you're away" |
| 8. Feather Controller | "**Camera+Images read/write:** Allows Feather to show you your camera and activity images. Additionally, Feather will allow you to request a snapshot." |
| 9. Heatworks model 1 | "**thermostat r/w:** Allows your Heatworks MODEL 1 water heater to go into vacation mode when your home is set to away" |
| **VC2: Describing *Away* as a property of the thermostat alone, rather than something that affects the entire smart home** | |
| 10. Gideon | "**away read/write:** Allows Gideon to read and update the Away state of your thermostat" |
| 11. Muzzley | "**away read/write:** Allows Muzzley to read and update the Away state of your thermostat" |
| 12. Keen home smart vent | "**away read/write:** Allows Smart vent to read the state of your Thermostat and change the state from Away to Home" |
| **VC3: Both VC1 and VC2** | |
| 13. WeMo | "**away read/write:** Allows your WeMo products to turn off when your Nest Thermostat is set to Away and on when set to Home." |
| 14. IFTTT thermostat service | "**thermostat read/write:** Now you can turn on Nest Thermostat Applets that monitor when you're home, away and when the temperature changes." |
| **VC4: Descriptions that do not relate to the permission** | |
| 15. IFTTT thermostat service | "**away read/write**: Now you can set your temperature or turn on the fan with Nest Thermostat Applets on IFTTT" |
| 16. Life360 | "**away read/write:** We need this permission to automatically turn on/off your nest system" |

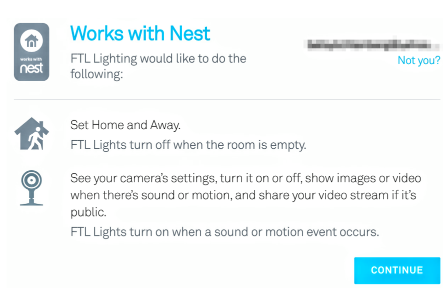


Fig. 5. An example from the Nest documentation on OAuth authorization [34] that displays a permission description violation (specifically, VC1) for the *Away r/w* and *Camera + images r/w* permissions.

**Finding 6. A significant number of apps provide incorrect permission descriptions, which may misinform users ($\mathcal{F}_6$).** As shown in Table 1, we found a total of 15 permission description violations in 13/39 apps from the $Apps_{nest}$ dataset. We classify these incorrect descriptions into four violation categories (*i.e.*, VC1 → VC4), based on the specific manner in which they misinform the user, such as requesting more privileges than required for the described need (*e.g.*, read/write permissions when only reading is required), or misrepresenting the effect of the use of the permission (*e.g.*, stating *Away* as affecting only the thermostat). That is, *over 33.33% of the apps we could integrate have violating permission descriptions.*

**Finding 7. In most cases of violations, apps request read/write permissions instead of read ($\mathcal{F}_7$).** In nine cases, apps request the more privileged *read/write* version of the permission, when they should have clearly requested the *read* version, as per their permission description (*i.e.,* VC1 in Table 1). For example, consider the "MyQ Chamberlain" app (Table 1, entry 3), which asks for the *thermostat read/write* permission, but whose description only suggests the need for the *thermostat read* permission, *i.e.,* "Allows Chamberlain to display your Nest Thermostat temperature in the MyQ app". More importantly, a majority of the violations of this kind occur for the *Away read/write* and *Camera+Images read/write* permissions, which may have serious consequences if these overprivileged apps are compromised, *i.e.,* as *Away read/write* regulates control over indicating whether a user is at home or out of the house, and *Camera+Images read/write* may allow apps to turn off the Nest cam via the *is_streaming* variable. These violations exist in spite of Nest guidelines that mention the following as a *Key Point*: *"Choose 'read' permissions when your product needs to check status. Choose 'read/write' permissions to get status checks and to write data values."* [33]. Finally, we found that the *Nest documentation may itself have incorrect instructions*, *e.g.,* the Nest's documentation on OAuth 2.0 authentication [34] shows an example permission prompt that incorrectly requests the *Away read/write* permission while only needing read access, *i.e.,* with the description "FTL Lights turn off when the room is empty", as shown in the Figure 5.

**Finding 8. The Nest product review is insufficient when it comes to reviewing the correctness of permission descriptions and requests by apps ($\mathcal{F}_8$).** The Nest product review suggests the following two rules, violating which may cause apps to be rejected: (1) *"3.3. Products with names, descriptions, or permissions not relevant to the functionality of the product"*, and (2) *"3.5. Products that have permissions that don't match the functionality offered by the products"* [35]. Our findings demonstrate that the 16 violations discovered violate either one or both of these rules (*e.g.,* by requesting read/write permissions, when the app only requires read). The fact that the apps are still available suggests that the Nest product review may not be rigorously enforced.

**Finding 9. Apps often incorrectly describe the *Away* field as a local field of the Nest thermostat, which is misleading ($\mathcal{F}_9$).** One example of this kind (VC2 in Table 1) is the *Keen Home* app described in Section 3.2 (Table 1. entry 12), which states that it needs *Away read/write* in order to *"Allow Smart vent to read the state of your Thermostat and change the state from Away to Home"*. As a result, *Keen Home* misrepresents the effect and significance of writing to the *Away* field, by making it seem like *Away* is a variable of the thermostat, instead of a field that affects numerous devices in the home. Gideon and Muzzley (entries 10 and 11 in Table 1) exhibit a similar anomaly. Our hypothesis is that such violations occur because Nest originally started as a smart thermostat that gradually evolved into a smart home platform. Finally, in addition to misleading descriptions classified as VC1 and VC2, we discovered apps whose permission descriptions did not relate to the permissions requested (VC4), and apps whose descriptions satisfied both VC1 and VC2 (VC3).

The accuracy of permission descriptions is important, as the user has no other source of information upon which to base their decision to trust an app. Nest recognizes this, and hence, makes permissions and descriptions a part of its product review. The discovery of inaccurate descriptions not only demonstrates that apps may be overprivileged, but also that Nest's design review process is incomplete, as it puts all its importance on getting the user's consent via permission prompts (*e.g.,* in Findings 5→9), but not on what information is actually shown.

*3.3.4 Application SSL Use.* The previous section demonstrated that smart home apps may be overprivileged in spite of a dedicated product review. An adversary may be able to compromise the smart home by exploiting vulnerabilities in such overprivileged apps. Thus, we decided to empirically derive an estimate of how vulnerable smart home apps are in terms of their use of SSL.

We used two datasets for this experiment, *i.e.,* the $\text{Apps}_{general}$ dataset consisting of 650 generic smart home (Android) apps crawled from Google Play, and an extended version of the $\text{Apps}_{nest}$ dataset, *i.e.,* the $\text{Apps}_{nestExt}$ dataset, which consists of 111 Android apps built for Works with Nest devices (*i.e.,* including the ones for which we do not possess devices). We analyzed each app from both the datasets using MalloDroid [15], to discover common SSL flaws.

**Finding 10. A significant percentage of general smart home management apps, as well as apps that connect to Nest have serious SSL vulnerabilities ($\mathcal{F}_{10}$).** 20.61% (*i.e.,* 134/650) of the smart home apps from the $\text{Apps}_{general}$ dataset, and 19.82% (*i.e.,* 22/111) apps from the $\text{Apps}_{nestExt}$ dataset, have at least one SSL violation as flagged by MalloDroid. Specifically, in the $\text{Apps}_{nestExt}$ dataset, the most common cause of an SSL vulnerability is a broken *TrustManager* that accepts *all certificates* (*i.e.,* 20 violations), followed by a broken *HostNameVerifier* that does not verify the hostname of a valid certificate (*i.e.,* 11 violations). What is particularly worrisome is that apps such as *MyQ Chamberlain* and *Wemo* have multiple SSL vulnerabilities as well as the *Away read/write* permission. Next, we demonstrate an end-to-end attack on the Nest security camera, using one of the SSL vulnerabilities discovered from this analysis, and the Nest*Away read/write* permission.

## 4 SIGNIFICANCE OF THE FINDINGS AND CHALLENGES FOR FUTURE RESEARCH

Our findings ($\mathcal{F}_1 \rightarrow \mathcal{F}_{10}$) expose critical gaps in the security of popular data store-based smart home platforms. Moreover, we demonstrate that these findings can also be combined to form an instance of a lateral privilege escalation attack [45], in the context of smart homes (Section 4.1). Moreover, we have reported the vulnerabilities that allow us to execute this attack, and summarize the current status of each of the findings (Section 4.2). Finally, given that platforms are becoming increasingly closed, and that our analysis relies on several key platform and API characteristics, we discuss the challenges that future research will face in emulating our methods (Section 4.3).

### 4.1 Lateral Privilege Escalation

While our findings from the previous sections are individually significant, we demonstrate that they can be combined to form an instance of a lateral privilege escalation attack [45], in the context of smart homes. That is, we demonstrate how *an adversary can compromise one product (device/app) integrated into a smart home, and escalate privileges to perform protected operations on another product, leveraging routines configured via the centralized data store.*

This attack is interesting in the context of smart homes, because of two core assumptions that it relies on (1) low-integrity (or non-security) smart home products may be easier to directly compromise than high-integrity devices such as the Nest Cam (*i.e.,* none of the SSL vulnerabilities in $\mathcal{F}_{10}$ were in security-sensitive apps), and (2) while low-integrity devices may not be able to directly modify the state of high-integrity devices ($\mathcal{F}_1$), they may be able to indirectly do so via *automated routines* triggered by global smart home variables ($\mathcal{F}_4$). (3) Moreover, since the low-integrity device is not being intentionally malicious, but is compromised, the product review process would not be useful, even if it was effective (which it is not, as demonstrated by $\mathcal{F}_5 \rightarrow \mathcal{F}_9$). This last point distinguishes a lateral privilege escalation from actions of malicious apps that trigger routines (*e.g.,* the "fake alarm attack" discussed in prior work [18]). These conditions make lateral privilege escalation particularly interesting in the context of smart home platforms.

**Attack Scenario and Threat Model:** We consider a common man-in-the-middle (MiTM) scenario, similar to the SSL-exploitation scenarios that motivate prior work [15, 48]. Consider Alice, a smart home user who has configured a security camera to record when she is away (*i.e.,* using the *away* variable in the centralized data store). Bob is an acquaintance (*e.g.,* a disgruntled employee or an ex-boyfriend) whose motive is to steal a valuable from Alice's house without being recorded by the camera. We assume that Bob also knows that Alice uses a smart switch in her home, and controls it via its app, which is integrated with Alice's smart home. Bob follows Alice, and connects to the same public network (*e.g.,* a coffee shop), sniffs the access token sent by the switch's app to its server using a known SSL vulnerability in the app, and then uses the token to directly control the *away* variable. Setting the *away* to "home" confuses the security camera into thinking that Alice is at home, and it stops recording. Bob can now burglarize the house without being recorded.

**The Attack:** The example scenario described previously can be executed on a Nest smart home, using the Nest Cam and the TP Link Kasa switch (and the accompanying Kasa app). We compromise the SSL connection of Kasa app, which was found to contain a broken SSL TrustManager in our analysis described in Section 3.3. We choose Kasa app as it requests the sensitive *Away read/write* permission, and has a sizable user base (1M+ downloads on Google Play [5]). It is interesting to note that the Kasa app has also passed the Nest product review and is advertised on the Works with Nest website [36]. We use *bettercap* [4] as a MiTM proxy to intercept and modify unencrypted data. Additionally, as described in the attack scenario, we assume that (1) the victim's Nest smart home has the Nest Cam and the Kasa switch installed, (2) the popular routine which triggers the Nest Cam to stop recording when the user is home is enabled, and (3) the user connects her smartphone to a network to which the attacker has access (e.g., coffee shop, office).

The attack proceeds as follows: (1) The user utilizes the Kasa app to control the switch, while the user's mobile device is connected to public network. (2) The attacker uses a MiTM proxy to intercept Kasa app's attempt to contact its own server, and supplies the attacker's certificate to the app during the SSL handshake, which is accepted by the Kasa app due to the faulty TrustManager. (3) The Kasa app then sends an authorization token to the MiTM proxy (*i.e.,* assuming it is the authenticated server), which is stolen by the attacker. This token authorizes a particular client app to send commands to the TP Link server. (4) Using the stolen token, the attacker instructs the TP Link server to set the smart home's *away* variable to "home", while the user is actually "away". This action is possible as the TP Link server (*i.e.,* Web app) has the *-Away read/write* permission for the user's Nest smart home. (5) This triggers the routine in the Nest Cam, which stops recording.

It should be noted that while this is one verified instance of a lateral privilege escalation attack on DSB smart home platforms, given the broad attack surface indicated by our findings, it is likely that similar undiscovered attacks exist.

### 4.2 Current Status of Vulnerabilities

We reported the discovered vulnerabilities to Philips ($\mathcal{F}_2$, $\mathcal{F}_3$), Nest/Google ($\mathcal{F}_4 \rightarrow \mathcal{F}_{10}$), and TP Link ($\mathcal{F}_{10}$) in mid-2018. Since then, vendors have responded, confirmed our findings, and even deployed fixes. This section summarizes the current status of the vulnerabilities, backed by additional experimental analysis where needed. A detailed account of our reporting experience with vendors can be found in our online appendix [1].

*4.2.1 Status of Permission Enforcement Vulnerabilities in Hue.* We reported findings $\mathcal{F}_2$ and $\mathcal{F}_3$ to Philips Lighting (*i.e.,* the owner of the Hue brand) along with a proof-of-concept script demonstrating the attacks. Hue confirmed our findings, and informed us that the latest release version 1931069120 mitigates these vulnerabilities.

**Experimental evaluation methodology:** We *experimentally evaluated Hue's claim*, using two kinds of third-party apps allowed on Hue, *i.e., local* and *cloud* apps. To elaborate, our exploits demonstrated in Section 3.1 ($\mathcal{F}_2$ and $\mathcal{F}_3$) can be executed from a *local app*, *i.e.,* a third-party app installed on a device connected to the same local network as the Hue bridge. However, Hue supports

[5] https://play.google.com/store/apps/details?id=com.tplink.kasa_android

another kind of third-party app, a *cloud app*, which uses the Hue remote API to remotely issue commands to the lights, and unlike local apps, does not need to be connected to the local network. Note that the *cloud app* needs a local app to act as its *proxy* on the hub, which has to be registered by using the access token for the remote app to issue a linkbutton=true command to the hue endpoint URL,[6] simulating the button-press on the Hue bridge.

We first used a local app, wmlocalapp (created in Section 3.1), to test whether our exploits for $\mathcal{F}_2$ and $\mathcal{F}_3$ still worked on the new Hue local API. We then created a cloud app named wmremoteapp in Hue's developer portal, to test our exploits for $\mathcal{F}_2$ and $\mathcal{F}_3$ in a scenario where the attacker controls a cloud app. Our wmremoteapp was instantly approved after submission to Hue, which indicates that it may not have undergone a thorough product review. Additionally, we created and registered wmlocalproxyapp, a local app that would act as a proxy for the cloud app on the Hue bridge.

**Key results:** We confirmed that $\mathcal{F}_3$ *no longer affects Hue*, *i.e.*, the new version prevents apps from deleting other apps from the whitelist. Further reverse-engineering revealed that Hue enforces this policy by obfuscating the *application-key* of the apps in the whitelist section of the data store. That is, apps cannot delete what they cannot address. However, this also means that the effectiveness of the mitigation relies on the complexity of the obfuscation; it will be invalid once an adversary devises a way to generate obfuscated names from arbitrary application metadata. An access control policy for the whitelist would be a more direct solution to this problem. Further, we discovered that $\mathcal{F}_2$ *still holds* if the attacker uses a Hue cloud app, and even bypasses Hue's product review-based defenses, leading to the following finding:

**Finding 11. *Cloud* apps can bypass user consent repeatedly ($\mathcal{F}_{11}$).** Since a cloud app must have an accompanying local proxy app to execute commands using the Hue bridge, it is reasonable to allow cloud apps to modify the *linkbutton* and add their proxy remotely. However, in our experiments, we discovered that wmcloudapp could modify *linkbutton repeatedly*, and thus, register *multiple local apps*. Moreover, local apps are not bound to the remote app that installed them, and hence, would persist even after the user removed the misbehaving wmcloudapp. The most important facet of this problem is that our misbehaving wmcloudapp is registered with Hue, and hence it would have been possible for users to install it.[7]

*4.2.2 SSL Vulnerability in TP Link's KASA.* We reported the details of the SSL vulnerability exploited in Section 4.1 to TP-Link, who acknowledged the issue and resolved it to a bug in the Android 4.x compatibility library. While TP Link did not elaborate on the exact part of the library that was problematic, they stated that future updates of the Kasa app would contain a fix. We statically and dynamically analyzed the most recent version of the Kasa app (version 2.13.0.858), and confirmed that (1) the vulnerable lines of code (*i.e.,* a TrustManager that accepts all certificates) were still present, however, (2) they were not being used for SSL connections, as our dynamic MiTM attack (Section 4.1) did not work.

*4.2.3 Status of vulnerable Nest routines, and misinformation in third-party Works with Nest apps.* We reported vulnerabilities in Nest in two distinct sets, (1) Report 1, describing the vulnerability of security-sensitive Nest devices to lateral privilege escalation, via routines ($\mathcal{F}_4$), and (2) Report 2, describing the inconsistent prompts, permission descriptions, and SSL misuse in third-party Works with Nest apps ( $\mathcal{F}_6$,$\mathcal{F}_7$, $\mathcal{F}_9$, $\mathcal{F}_{10}$), as well as the problems in Nest's product review process ($\mathcal{F}_5$ and $\mathcal{F}_8$). As Nest was non-responsive,[8] we submitted the two reports to Google through their bug reporting system.[9]

[6]https://api.meethue.com/bridge/0/config
[7]We ensured that wmcloudapp was clearly marked as a test app, and that no real user installed it during our experiments.
[8]Nest operated independently from Google from 2015-2018, and hence, reporting to Google was a non-obvious step in 2018.
[9]https://www.google.com/appserve/security-bugs/m2/new

**Current Status of** Report 1**:** Initially, Google's position was that the lateral privilege escalation was purely due to the SSL vulnerability in TP Link's KASA app. As we explained how routines in Nest were key for the attack, the engineers assigned to the bug report acknowledged the existence of a design-level flaw in Nest routines. However, this flaw is extremely hard to fix without disabling sensitive routines (*e.g.,* the security camera routine in Nest), or employing integrity checks in the platform's architecture (see Section 5 for additional discussion), and hence, currently remains exploitable (although the immediate attack vector, the Kasa app, is fixed). A detailed account of this discussion is available in our online appendix [1].

**Current Status of** Report 2**:** As of today, most of the issues described in this report remain unaddressed, *including the instances of misinformation in the Nest documentation itself* (see $\mathcal{F}_7$). Google's position was that the onus of fixing these issues in apps was on the third-party app developers to review the permissions that their apps request, and hence, that the findings should be disclosed on developer forums. Note that all the reported apps had undergone direct scrutiny from Nest through their review process and passed that process before deployment to the end-user. This exchange brings up a crucial question, for situations where the platform may not be willing to even address the over-privilege in existing apps when reported by consumers or researchers: *Who should the end-user should deem liable in the instance of a security incidence involving an overprivileged smart home app; the app developer, or the platform that vetted the app and allowed users to install it?*

### 4.3 The Feasibility of Analyzing Evolving Smart Home Platforms

The market for smart home products and platforms is now reaching a critical mass in terms of consumer adoption. This has resulted in an ecosystem of rapidly evolving and fragmented platforms. As of now, we do not have a concrete understanding of how platform evolution helps, or hurts, the *applicability* of existing security analysis approaches. Acquiring such an understanding would be instrumental in helping future security researchers recognize the opportunities as well as challenges posed by evolving characteristics of smart home platforms.

We pose a seemingly simple but nuanced research question: *How feasible would the analysis performed in this paper be on smart home platforms in the near future?* To address this question, we (1) identify the essential, platform-independent properties that facilitate the security analyses explored in this paper, and (2) evaluate six additional platforms to understand if they exhibit these properties. We conclude the section by identifying the foremost challenge for similar research in the future, drawing from the evidence obtained in our evaluation.

*4.3.1 Platform-independent* essential properties. The security evaluation performed in Section 3 can be categorized into five independent analyses: ($\mathcal{A}_1$) an analysis of platform *permission enforcement*, ($\mathcal{A}_2$) the accuracy of *install-time permission descriptions*, ($\mathcal{A}_3$) the accuracy of *runtime permission prompts* , ($\mathcal{A}_4$) the security *impact of routines*, and ($\mathcal{A}_5$) *SSL misuse* by third party mobile apps. We identify the five platform-independent ***essential properties*** that facilitate these analyses:

**Property 1 - ($\mathcal{P}_1$): Availability of public API access to test permission enforcement.** In order to test whether the purported permission enforcement mechanisms that exist in a given platform function properly in practice, it is necessary to have access to public facing platform APIs that would enable us to generate permission maps via automated testing (*i.e.,* ***for*** $\mathcal{A}_1$, Section 3.1).

**Property 2 - ($\mathcal{P}_2$) Platform-mandated *third-party-specified* permission descriptions.** Smart home platforms generally inform users about the effect of platform permissions (*e.g.,* that the home/away r/w permission can "Set Home and Away", as seen in Figure 5). However, some platforms (*e.g.,* Nest) may also require developers to provide additional context to the user, via *install-time permission descriptions* describing *why* their app needs a particular permission. The

Table 2. Feasibility of Analyses $\mathcal{A}_1 \rightarrow \mathcal{A}_5$ on various smart home platforms.

| Analysis | Nest v1 | Nest v2 | SmartThings Classic | SmartThings v2 | HomeKit | Home Assistant | OpenHAB |
|---|---|---|---|---|---|---|---|
| $\mathcal{A}_1$: Permission Enforcement | ✓ | × | ✓ | ✓ | ✓ | ✓ | × |
| $\mathcal{A}_2$: Permission Description Accuracy | ✓ | × | × | × | ✓ | × | × |
| $\mathcal{A}_3$: Permission Prompt Accuracy | ✓ | × | × | × | × | × | × |
| $\mathcal{A}_4$: Impact of Routines | ⊘ | ⊘ | ✓ | ⊘ | ⊘ | ✓ | ✓ |
| $\mathcal{A}_5$: SSL Misuse in Third Party Apps | ✓ | ⊘ | ✓ | ⊘ | ✓ | ✓ | ✓ |

✓= feasible, × = not feasible, and ⊘ = partially feasible

availability of such descriptions is critical for understanding how applications may misinform users about their actual intent, and violate platform design policies (*i.e., **for*** $\mathcal{A}_2$, Section 3.3.1).

**Property 3 - ($\mathcal{P}_3$) Platform-mandated *third-party-specified* runtime permission prompts.** In addition to install-time descriptions, platforms may also require applications to use run-time prompts before performing a sensitive action (*e.g.,* as Nest does for home/away), thereby allowing the user to make a more informed decision. These prompts are necessary to understand if a third-party application's actual use of a permission is valid (*i.e., **for*** $\mathcal{A}_3$, Section 3.2).

**Property 4 - ($\mathcal{P}_4$) Published third-party routines for home automation.** Routines or automations are generally supported by platforms through third-party integrations, *i.e.,* by integrating devices directly via Zigbee or Z-wave, or indirectly by provisioning API access to third-parties, or through third-party *IoT apps* hosted on the platform itself (*e.g.,* SmartThings SmartApps). As routines may be exploited by attackers, the availability of third-party routines is critical for assessing the presence or prevalence of vulnerabilities that would facilitate attacks such as the lateral privilege escalation attack explored in this paper (*i.e., **for*** $\mathcal{A}_4$, Sections 3.2 and 4.1).

**Property 5 - ($\mathcal{P}_5$) Availability of third-party mobile applications.** Smart home platforms are inextricably tied to mobile apps that provide users with a convenient means of controlling various aspects of their smart home, and even facilitate routines (*e.g.,* Yeti [65] and Yonomi [66]). The availability of mobile apps is not only needed for analyzing the security of the communications used by the apps themselves (*i.e., **for*** $\mathcal{A}_5$, Section 3.3.4), but also for understanding the use of permission descriptions and prompts by third-parties (*i.e., **for*** $\mathcal{A}_2 \rightarrow \mathcal{A}_3$, Sections 3.2 and 3.3.1).

*4.3.2 Evaluation of 6 Additional Smart Home Platforms.* We analyzed six smart home platforms (in addition to Nest and Hue) for the presence of properties $\mathcal{P}_1 \rightarrow \mathcal{P}_5$, in order to understand the feasibility of performing $\mathcal{A}_1 \rightarrow \mathcal{A}_5$ on them. Table 2 summarizes the results of this feasibility analysis. We now provide a brief overview of our general empirical evaluation methodology, followed by the results of the feasibility analysis for each platform.

**General Evaluation Methodology:** We followed a systematic, 4-step methodology for the feasibility analysis: **(1) *Platform Selection.*** We selected six platforms from popular publicly available smart home platforms, based on one foundational trait that precedes $\mathcal{P}_1 \rightarrow \mathcal{P}_5$: *allowing the integration* of third-party routines, mobile apps, and devices. **(2) *Testing for Public APIs.*** For each platform, we then determined the availability of *public* APIs from all available sources (*e.g.,* documentation, official website). If we could register as a developer with the platform, acquire an API key, and make API calls to access platform resources, we considered $\mathcal{P}_1$ satisfied (*i.e.,* conversely, platforms that allowed API access to a limited/closed set of partners did not satisfy $\mathcal{P}_1$). **(3) *Analyzing Permission Models.*** We examined the provided developer documentation to extract the permission model, and to determine if developers were required to specify custom install-time permission descriptions ($\mathcal{P}_2$) and runtime prompts ($\mathcal{P}_3$) to provide users with more context. Moreover, we examined whether the prompts could be *programmatically triggered* for analysis through integration of our own test app/device to the platform (*i.e.,* and hence, tested the *extent* to which $\mathcal{P}_3$ was satisfied). **(4) *Mining third-party clients.*** We tried to acquire artifacts that represent routines, such as IoT apps published in markets (*e.g.,* the SmartThings public repo [52]), descriptions of automation in text-form on the platform's website (*e.g.,* the Works with Nest website [38]), or automations enabled by third-party mobile apps integrated with the platform. Aside from testing for $\mathcal{P}_4$, this step also allowed us to test for $\mathcal{P}_5$ (*i.e.,* as we searched for mobile apps as well).

We carefully considered platform-specific nuances when executing Steps 1→4, and ***experimentally confirmed our claims for all platforms***. The rest of this section provides a brief overview of each analyzed platform, followed by a summary of our analysis results.

**1. Feasibility Evaluation of Nest v2:** Google is closing its *Works with Nest* platform on August 31, 2019 in favor of a more tightly-integrated *Works with Google Assistant* platform.[10] We term this new platform Nest v2, while the version we analyzed in this work is termed as Nest v1. The transition from v1 to v2 alters the fundamental nature of Nest, from an open, decentralized, platform to a relatively closed platform (*i.e.,* with API access to select vendors) built around the Google Assistant.

From our analysis, we conclude that the *closed* nature of Nest v2 violates most of the properties, rendering corresponding analyses performed in this paper infeasible. For instance, the ability of researchers to access the API in Nest v2 will be constrained, as the platform is geared towards helping vendors integrate their devices or products with Google Assistant. At most, researchers will be able to create their own virtual device and an interface for Google Assistant to access that device (*i.e.,* unlike Nest v1, which has a general-purpose public API that can be used to access multiple *other* devices and resources). Thus, Nest v2 violates $\mathcal{P}_1$, making $\mathcal{A}_1$ infeasible.

Further, Nest v2 does not require developers to write custom permission descriptions or prompt the user before using a permission, as *permissions are acquired by Google Assistant when integrating the device with the platform*. Hence, Nest v2 loses the context of requiring/using permissions, violates $\mathcal{P}_2$ and $\mathcal{P}_3$, and invalidates $\mathcal{A}_2$ and $\mathcal{A}_3$. Moreover, routines will only be created and managed via Google Assistant, which means that no repositories of routines will be available, requiring researchers to analyze *potential* routines (*e.g.,* from integrations described on the Works with Google Assistant website). Hence, $\mathcal{P}_4$ is partially satisfied, and it may be somewhat feasible to perform $\mathcal{A}_4$, although incredibly difficult to do so with completeness or at scale. Finally, since the Google Home mobile app is the only official way for the user to access the platform, we do not foresee the development of third-party mobile apps that integrate with Nest v2. However, it is common for vendors to provide mobile apps as alternate mediums to control their devices, and since some vendors are being tightly integrated to Nest v2 after a thorough review process [11], Nest v2 may potentially partially satisfy $\mathcal{P}_5$, and hence, $\mathcal{A}_5$.

**2. Feasibility Evaluation of SmartThings Classic and v2:** A particularly interesting aspect of SmartThings is that it allows developers to publish Groovy-based IoT apps (*i.e.,* called SmartApps) in a platform-provided market. This existing SmartThings "Classic" platform is now being phased out in favor of the new SmartThings v2 platform[12] launched on March 18, 2018 that drastically deviates from this characteristic, *i.e.,* SmartThings v2 has eliminated Groovy-based SmartApps. Instead, SmartApps are now manifested as Web hook endpoints [51] or AWS Lambda functions [50].

Our analysis confirms that all properties except $\mathcal{P}_2$ and $\mathcal{P}_3$ hold for SmartThings Classic (*i.e.,* as SmartThings does not mandate developer-specified permission descriptions or prompts). Hence, a majority of our analyses are feasible on the classic version (*i.e.,*$\mathcal{A}_1$, $\mathcal{A}_4$, and $\mathcal{A}_5$). However, the changes in SmartThings v2 make $\mathcal{A}_4$ and $\mathcal{A}_5$ partially infeasible. Specifically, $\mathcal{P}_4$ is affected due to the lack of centrally published and hosted SmartApps in SmartThings v2. That is, as SmartApps will be reduced to remote endpoints, researchers will only have text descriptions of routines from

[10] https://blog.google/products/google-nest/helpful-home/
[11] https://www.blog.google/products/google-nest/updates-works-with-nest/
[12] https://blog.smartthings.com/news/smartthings-updates/the-new-smartthings-app-is-here/

vendor websites to analyze, rendering $\mathcal{A}_4$ partially feasible. Similarly, while third-party mobile app integration is technically possible, it is currently unavailable, which means that $\mathcal{P}_5$ does not fully hold, and performing $\mathcal{A}_5$ would be infeasible at least in the near future.

**3. Feasibility Evaluation of HomeKit:** Apple HomeKit [24] is a proprietary framework that allows interaction among different devices (called *accessories*) in the home through iOS apps. Once accessories are integrated into the framework, users can remotely automate them via iOS apps.

While HomeKit is a closed platform similar to Nest v2, it does provide hobbyists with API-support to explore/test the platform. This access would allow researchers to create their own accessories (*i.e.,* devices), while the typical iOS testing and development tools may be used for analyzing the permission enforcement, and access to these devices (*i.e.,* fully satisfying $\mathcal{P}_1$ and facilitating $\mathcal{A}_1$). Further, developers are required to specify "usage descriptions", which is why $\mathcal{P}_2$ holds, facilitating $\mathcal{A}_2$. However, $\mathcal{A}_3$ is not applicable as there are no mandated prompts. Routines are not available in one place, but can be acquired by analyzing the Home app, *i.e.,* $\mathcal{P}_4$ partially holds, and hence $\mathcal{A}_4$ is partially feasible. Finally, as mobile apps are integral to this model, $\mathcal{A}_5$ is feasible.

**4. Feasibility Evaluation of Home Assistant:** Home Assistant [22] is an open-source framework for smart home management. Unlike other proprietary platforms which rely on the cloud, Home Assistant provides the option of hosting the automation server locally.

Home Assistant's open nature provides valuable opportunities for analysis. For instance, it is open source, allowing researchers to build it locally and automate the creation of the permission map as we discovered in our initial exploration (*i.e.,* $\mathcal{A}_1$ is feasible). Note that Home Assistant does not enforce device-level permissions, but instead, enforces access control among multiple users (*i.e.,* hence, the scope of the permission map changes). Publicly available automations [23] satisfy $\mathcal{P}_4$ and facilitate $\mathcal{A}_4$. Similarly, third-party apps for Home Assistant are not numerous, but exist, satisfying $\mathcal{P}_5$ and facilitating $\mathcal{A}_5$. However, Home Assistant does not exhibit $\mathcal{P}_2$ and $\mathcal{P}_3$ due to its unique permission model, *i.e.,* the user can directly define centrally managed groups that have a specific access to certain smart home resources, which precludes permission descriptions or prompts.

**5. Feasibility Evaluation of OpenHAB:** OpenHAB is an open-source framework that users can host locally or on the OpenHAB cloud service. Devices (*i.e., things*) are integrated with OpenHAB via *bindings* (*i.e.,* similar to device handlers in SmartThings). Users can then leverage these integrated things to create routines (called rules).

While OpenHAB provides bindings for various communication protocols, there is no permission system in place to connect third-party services. Thus, $\mathcal{P}_1 \rightarrow \mathcal{P}_3$ do not hold for OpenHAB, and $\mathcal{A}_1 \rightarrow \mathcal{A}_3$ are not applicable. However, as a significant number of OpenHAB rules (*i.e.,* routines) can be found in dedicated forums,[13] $\mathcal{P}_4$ holds, and $\mathcal{A}_4$ is feasible. Similarly, because there are third party apps that interface with this platform ($\mathcal{P}_5$) an app-based analysis ($\mathcal{A}_5$) is possible.

## 4.4 The challenge for future security research

Our feasibility evaluation reveals several interesting aspects of the smart home ecosystem. For example, some platforms such as Apple HomeKit, Home Assistant, and OpenHAB do not implement permissions at the granularity of a device (*i.e.,* instead, only implement multi-user separation, which is further absent in OpenHAB), a coarse-grained model that would be trivial to exploit once an authorization token is stolen (*i.e.,* even without a transitive exploit such as a lateral privilege escalation). More importantly, it demonstrates alarming trends for future research in this area. That is, none of the platforms are amenable to all of $\mathcal{A}_1 \rightarrow \mathcal{A}_5$, even partially (*i.e.,* except Nest v1 which was the initial focus of this paper). More importantly, we see that as platforms evolve, they become

[13] https://community.openhab.org/c/tutorials-examples

less open and transparent to introspection by security researchers. For instance, SmartApps in SmartThings v2 are hidden behind endpoints on the Web, and no longer as open to scrutiny as those in SmartThings v1. Similarly, Nest v2 abstracts platform API, routines, and most functionality behind the Google Assistant API, which is not public and only available to certain certified partners. This is in complete contrast with the publicly accessible API of Nest v1 that enabled the analysis in this paper. To continue investigating the security of smart home platforms, researchers must overcome the overwhelming ***challenge of*** (1) ***identifying and mining novel sources of routines and apps at scale, and*** (2) ***developing alternate methods of accessing platform APIs, which includes engaging platform vendors to acquire official API access.***

## 5 POTENTIAL DEFENSES

Of the 11 findings of this study, several could be *individually* prevented through well-known system-level defenses, application analysis techniques, or best-practices for developing secure software. However, the combination of these findings, in a lateral privilege escalation attack, would be hard to defend against, given the lack of integrity guarantees on smart home platforms. This section discusses the following potential defenses, as well as the challenge of securing the system against privilege escalation:

**1. Automated testing for permission enforcement:** Findings $\mathcal{F}_2$, $\mathcal{F}_3$, and $\mathcal{F}_{11}$ demonstrate the fundamental lack of correctness in permission enforcement in Hue. This gap can be mitigated using systematic approaches that test the platform-level permission enforcement by building and validating permission maps before system deployment [16].

**2. Bolstering platform reviews with text analytics:** While platforms often review third-party applications as the first line of defense, the coarse-grained review performed by Nest is susceptible to abuse by apps. That is, while Nest mandates both install-time permission descriptions as well as runtime prompts for sensitive actions, it *does not regulate the correctness of the text in the prompts/descriptions* ($\mathcal{F}_5 \rightarrow \mathcal{F}_9$). To address this gap, platforms could augment their product reviews with natural language processing (NLP) techniques to analyze the consistency of application-provided permission descriptions with the actual permissions requested (e.g., Whyper [43] and SmartAuth [59]), and in turn, enable the user to provide informed consent.

**3. Enabling security best-practices at the application design stage:** SSL misuse by Android applications was first documented by Fahl et al. [15] in 2012, and has been heavily studied since then [2, 13, 27, 54]. However, Android applications are still susceptible to SSL misuse ($\mathcal{F}_{10}$), which weakens the security of the smart home. To conclusively address SSL misuse in applications, platforms must encourage (and sometimes even mandate) developers to use safe defaults, and provide API support for even safer use cases (e.g., SSL Pinning). In parallel, platforms must also deploy static and dynamic techniques that detect cryptographic misuse in applications [2, 13, 15, 27, 41, 47, 54], for preventing flaws in current applications from affecting the smart home.

**4. Providing integrity guarantees:** Our lateral privilege escalation attack in Section 4.1 was possible because in DSB platforms, applications can use variables such as home/away as gadgets to modify high-integrity devices (e.g., the security camera). Thus, this is an integrity problem, wherein the platform allows low-integrity and high-integrity applications/devices to share the same object. Transitive integrity enforcement, *i.e.,* where the platform implements an integrity lattice, and only allows interactions among devices/apps if the said interactions do not violate the lattice, would theoretically address this problem. However, in practice, enforcing integrity policies is challenging, given that variables such as home/away are used for both security and non-security purposes (e.g., to manipulate lights, as well as the camera). As a result, enforcing integrity without breaking functionality is an open research problem on platforms such as Nest.

## 6 RELATED WORK

Smart home platforms are an extension of the new modern OS paradigm, the security problems in smart home platforms are similar to prior modern OSes (*e.g.,* application over-privilege, incorrect enforcement). As a result, some of the same techniques may be applied in detecting such problems, as Alrawi et al. demonstrated in their analysis of network-facing smart home devices [3]. For example, our work uses automated testing to derive permission maps in a manner similar to Felt et al.'s seminal evaluation of Android permission enforcement [16]. We also leverage lessons from prior work on SSL misuse [15, 41, 48, 54] to perform the SSL Analysis (Section 3.3.4) and the MiTM exploit (Section 4.1). The lack of transitivity in access control that we observe is similar to prior observations on Android [9, 17, 30, 31]; however, the implications are different in the smart home. The novelty of this paper is rooted in using lessons learned from prior research in modern OS and application security to identify problems in popular but under-evaluated platforms such as Nest and Hue, and moreover, in demonstrating the potential misuse of home automation *routines* for performing lateral privilege escalation.

While prior work analyzes IoT apps to study the *potential* for misuse [12], this paper is the first to demonstrate an end-to-end lateral privilege escalation attack involving routines (Section 4.1). Our focus on adversarial misuse and a demonstrated end-to-end attack distinguishes this paper from closely-related work in smart home security, such as the evaluation of the SmartThings platform and its apps by Fernandez et al. [18], and systems such as IoTSAN [39], Soteria [7], IoTGuard [8], and iRuler [60] that detect the side-effects of the concurrent execution of Samsung's SmartApps. Aside from our 11 novel findings ($\mathcal{F}_1 \rightarrow \mathcal{F}_{11}$), the value of this paper is in its holistic evaluation of home automation security, which accounts for permission text artifacts, product review-based defenses, and the detrimental impact of platform evolution on the feasibility of analysis.

In a similar vein as this work, prior work by Surbatovich et al. [56] analyzed the security and privacy risks associated with IFTTT recipes, which are trigger-action programs similar to routines. The key difference is that Surbatovich et al. examines the safety of individual recipes, while our work explores routines that may be safe on their own (*e.g.,* when *home*, turn off the Nest Cam), but which may be used as gadgets by attackers to attack a high-integrity device from a low-integrity device. Our holistic analysis is complementary to such per-routine analysis. Similarly, our work is complementary to prior work that analyzes the security of individual devices [40, 49], or the correctness of routines in terms of representing the user's requirements [6, 10, 32, 42, 67].

Finally, prior work has proposed novel access control enhancements for smart home platforms, such as provenance systems (*e.g.,* ProvThings [61]), systems such as ContexIoT [26] that enable highly-contextual runtime prompts, or systems such as SmartAuth [59] that analyze the consistency of an app's text description with its code, which may alleviate the concerns raised in this paper. However, such systems will also become exceedingly difficult to design, evaluate, and deploy, due to the evolutionary trends in smart home platforms (Section 4.3).

## 7 LESSONS AND CONCLUSION

Our findings ($\mathcal{F}_1$)→($\mathcal{F}_{11}$) demonstrate numerous gaps in the security of DSB platforms. We now distill the core lessons from our security findings from Nest and Hue, as well as the feasibility analysis with 6 additional platforms.

**Lesson 1 :** *Seamless automation must be accompanied by strong integrity guarantees.* It is important to note that the attack described in Section 4.1 can not be addressed by reducing overprivilege or via product reviews, since none of the components of the attack are overprivileged (*i.e.,* including TP Link Kasa), and our findings demonstrate that the Nest product review is insufficient ($\mathcal{F}_5 \rightarrow \mathcal{F}_9$). The attack was possible due to the integrity-agnostic execution of routines in Nest ($\mathcal{F}_4$). To mitigate such attacks, platforms need information flow control (IFC) enforcement that ensures strong integrity guarantees [5], and future work may explore the complex challenges of (1) specifying integrity labels for diverse devices and (2) enforcing integrity constraints without sacrificing automation.

**Lesson 2:** *Nest Product Reviews would benefit from at least light-weight static analysis.* Our findings demonstrate numerous violations of the Nest design policies that should have been discovered during the product review. Moreover, the review guidelines also state that products that do not securely transmit tokens will be rejected [35], but our simple static analysis using MalloDroid discovered numerous SSL vulnerabilities in Nest apps ($\mathcal{F}_{10}$), of which one can be exploited (Section 4.1). We recommend the integration of light-weight tools such as MalloDroid in the review process.

**Lesson 3:** *The security of the smart home indirectly depends on the smart phone (apps).* Smartphone apps have been known to be susceptible to SSL misuse [15], among other security issues (*e.g.,* unprotected interfaces [9]). Thus, unprotected smartphone clients for smart home devices may enable the attacker to gain access to the smart home, and launch further attacks, as demonstrated in Section 4.1. Ensuring the security of smart phone apps is a hard problem, but future work may triage smartphone apps for security analyses based on the volume of smart home devices/platforms they integrate with, thereby, improving the apps that offer the widest possible attack surface.

**Lesson 4:** *Popular but simpler platforms need urgent attention.* The startling gaps in the access control of Hue demonstrate that the access control of other simple (*i.e.,* homogeneous) platforms may benefit from a similar holistic security analysis ($\mathcal{F}_2$, $\mathcal{F}_3$, $\mathcal{F}_{11}$).

**Lesson 5:** *New Analysis Methods are required as smart home platforms become more restrictive to integrations.* Our feasibility analysis in Section 4.3 demonstrates how popular smart home platforms are becoming less transparent, and more amenable to security analysis. While this tighter control can help to alleviate certain security problems such as public API misuse, or side-stepping review protocols, it also shifts more control into the hands of the platforms, making them more difficult to examine. Thus, new methods of analysis that work within the boundaries of modern platform restrictions are needed. For instance, acquiring and studying the security implications of the increasingly common user-driven routines [29] offers a potentially viable alternative to studying developer-provided IoT apps.

## REFERENCES


[1] 2018. Online Appendix. https://sites.google.com/view/smart-home-routines-analysis-2/home.

[2] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev).* 49–61.

[3] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *IEEE S&P.* 208–226.

[4] BetterCAP. Accessed June 2018. BetterCAP stable documentation. https://www.bettercap.org/legacy///.

[5] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems.* Technical Report MTR-3153. MITRE.

[6] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How Users Interpret Bugs in Trigger-Action Programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems.*

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC).* 147–158.

[8] Z. Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Proceedings of the NDSS 2019 Symposium.*

[9] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and D. Wagner. 2011. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services.*

[10] Fulvio Corno, Luigi De Russis, and Alberto Monge Roffarello. 2019. Empowering End Users in Debugging Trigger-Action Rules. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems.* 388:1–388:13.

[11] Ry Crist. Accessed September 2018. A smart home divided: Can it stand? https://www.cnet.com/news/a-smart-home-divided-can-it-stand/.

[12] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 832–846.
[13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.
[14] Engadget. Accessed June 2018. SmartThings shows off the ridiculous possibilities of its connected home system. https://www.engadget.com/2014/01/11/smartthings-labs/.
[15] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*.
[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[17] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*.
[18] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*. 636–654.
[19] Gartner. Accessed June 2018. Gartner Says 8.4 Billion Connected Things Will Be in Use in 2017, Up 31 Percent From 2016. https://www.gartner.com/newsroom/id/3598917.
[20] Google Play. Accessed June 2018. Nest. https://play.google.com/store/apps/details?id=com.nest.android.
[21] Google Play. Accessed June 2018. Philips Hue. https://play.google.com/store/apps/details?id=com.philips.lighting.hue2.
[22] Home Assistant. Accessed July 2019. https://www.home-assistant.io.
[23] Home Assistant Automations. Accessed July 2019. Cookbook. https://www.home-assistant.io/cookbook/.
[24] HomeKit. Accessed July 2019. Homekit helps to control devices using Home App. https://www.apple.com/ios/home/.
[25] IFTTT. Accessed June 2018. IFTTT helps your apps and devices work together. https://ifttt.com/.
[26] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z Morley Mao, Atul Prakash, and Shanghai JiaoTong Unviersity. 2017. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*.
[27] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. 2017. CogniCrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 931–936.
[28] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. 1994. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys (CSUR)* 26, 3 (Sept. 1994).
[29] Sunil Manandhar, Kevin Moran, Kaushal Kafle, Ruhao Tang, Denys Poshyvanyk, and Adwait Nadkarni. 2020. Towards a Natural Perspective of Smart Homes for Practical Security and Safety Analyses. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*.
[30] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *Proceedings of the 25th USENIX Security Symposium*.
[31] Adwait Nadkarni and William Enck. 2013. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[32] Chandrakana Nandi and Michael D Ernst. 2016. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 97–102.
[33] Nest Developers. Accessed June 2018. How to Choose Permissions and Write Descriptions. https://developers.nest.com/documentation/cloud/permissions-overview.
[34] Nest Developers. Accessed June 2018. OAuth 2.0 Authentication and Authorization. https://developers.nest.com/documentation/cloud/how-to-auth.
[35] Nest Developers. Accessed June 2018. Product Review Guidelines. https://developers.nest.com/documentation/cloud/product-review-guidelines.
[36] Nest Labs. Accessed June 2018. Kasa - Works with Nest Store. https://workswith.nest.com/company/tp-link-research-america-corp/kasa.
[37] Nest Labs. Accessed June 2018. Nest Developers. https://developers.nest.com///.
[38] Nest Labs. Accessed June 2018. Works with Nest. https://nest.com/works-with-nest//.
[39] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. 191–203.
[40] Sukhvir Notra, Muhammad Siddiqi, Hassan Habibi Gharakheili, Vijay Sivaraman, and Roksana Boreli. 2014. An experimental study of security and privacy risks with emerging household appliances. In *Proceedings of the 2014 IEEE Conference on Communications and Network Security (CNS)*. 79–84.

[41] Lucky Onwuzurike and Emiliano DeCristofaro. 2015. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless & Mobile Networks*. 15.
[42] Mitali Palekar, Earlence Fernandes, and Franziska Roesner. 2019. Analysis of the Susceptibility of Smart Home Programming Interfaces to End User Error. In *IEEE Workshop on the Internet of Safe Things (SafeThings) (SafeThings'19)*. ACM, New York, NY, USA.
[43] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the USENIX Security Symposium*.
[44] Philips. Accessed June 2018. Philips Hue: Personal Wireless Lighting. https://www2.meethue.com/en-us/about-hue.
[45] Dave Piscitello. 2016. What is Privilege Escalation. https://www.icann.org/news/blog/what-is-privilege-escalation.
[46] Popular Science. Accessed June 2018. Stop shouting at your smart home so much and set up multi-step routines. https://www.popsci.com/smart-home-routines-apple-google-amazon/.
[47] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.
[48] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. 2015. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*. 17–32.
[49] Vijay Sivaraman, Dominic Chan, Dylan Earl, and Roksana Boreli. 2016. Smart-phones attacking smart-homes. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 195–200.
[50] Smartthings Developers. Accessed July 2019. Hosting with AWS Lambda. https://smartthings.developer.samsung.com/docs/guides/smartapps/aws-lambda.html.
[51] Smartthings Developers. Accessed July 2019. Self-hosting (WebHook). https://smartthings.developer.samsung.com/docs/guides/smartapps/webhook-apps.html.
[52] Smartthings Developers. Accessed June 2018. Documentation. http://developer.smartthings.com/.
[53] SmartThings Support. Accessed June 2018. Routines in the SmartThings Classic app. https://support.smartthings.com/hc/en-us/articles/205380034-Routines-in-the-SmartThings-Classic-app.
[54] David Sounthiraraj, Justin Sahs, Zhiqiang Lin, Latifur Khan, and Garrett Greenwood. 2014. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*.
[55] Stringify. Accessed June 2018. Stringify | Change Your Life by Connecting Everything. https://www.stringify.com//.
[56] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web*. 1501–1510.
[57] TechCrunch. Accessed June 2018. Google Assistant is adding Routines and location-based reminders. https://techcrunch.com/2018/02/23/google-assistant-is-adding-routines-and-location-based-reminders/.
[58] The Verge. Accessed June 2018. You can soon activate multi-step routines in Alexa with a single command. https://www.theverge.com/2017/9/27/16375050/alexa-routines-echo-amazon-2017/.
[59] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Security Symposium*.
[60] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1439–1453.
[61] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and Logging in the Internet of Things. In *Network and Distributed Systems Symposium*.
[62] Works with Nest Store. Accessed June 2018. Keen Home. https://workswith.nest.com/company/leaksmart/leaksmart//.
[63] Works with Nest Store. Accessed June 2018. Nest x Yale Lock. https://workswith.nest.com/company/yale/nest-x-yale-lock//.
[64] Works with Nest Store. Accessed June 2018. Somfy Protect. https://workswith.nest.com/company/somfy-protect-by-myfox-sas/works-with-somfy-protect//.
[65] Yeti. Accessed June 2018. Yeti - Simplify the control of your smart home. https://getyeti.co/.
[66] Yonomi. Accessed June 2018. Yonomi app – Yonomi. https://www.yonomi.co.
[67] Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: Synthesizing and Repairing Trigger-action Programs Using LTL Properties. In *Proceedings of the 41st International Conference on Software Engineering*. 281–291.