The Security of Samsung's SmartThings Framework - Dangers and Chances

Dominik Delgado Steuter, Tom Retterath

July 18, 2022

Contents

1 Introduction

Smart home technology is the next big thing in the field of consumer-accessible technology. The security of such systems needs to be checked in order to prevent concerns like privacy-invasion or potential theft.

There has been research in the security of the protocols used in smart homes, e.g. ZWave. Individual apps have also been analyzed. More important however is the security of the programming framework in which the smart home apps are written in. It is hard to retroactively change design aspects of the programming framework. That is because all the smart home apps built with the framework are dependent on that version of the framework. This expounds the importance of a robust framework to begin with.

Earlence Fernandes et al. [1] devoted themselves to the question whether or not the currently used programming framework is resistant enough against attacks. They published the first ever study on the security of the underlying framework of current smart home apps. The goal of their study was to prevent security concerns in smart home technology before the technology is too widespread to change it.

For that reason they analyzed the currently most popular smart home technology framework: Samsung's SmartThings. They view it as a good representative of other frameworks which might emerge in the future.

Abstractly saying, the framework (in this case SmartThings) is the environment in which the smart home apps, the smart devices and the executive back end are communicating which one another. Details to this will follow in Section 2.1.

Their studies revealed that most of the apps in the framework are majorly overprivileged as a result of the framework's design. Attackers can easily abuse this fact and are able to cause moderate up to increased harm to the victim. This is proven in the study by multiple attack demonstrations. An overview is given in Section 3.4.

For instance, a custom made smart app could present itself as a battery status monitor for smart devices while actually snooping on door lock codes. Attackers would be able enter a previously locked house.

2 Background

2.1 The SmartThings Framework

2.1.1 The General Structure of the Framework

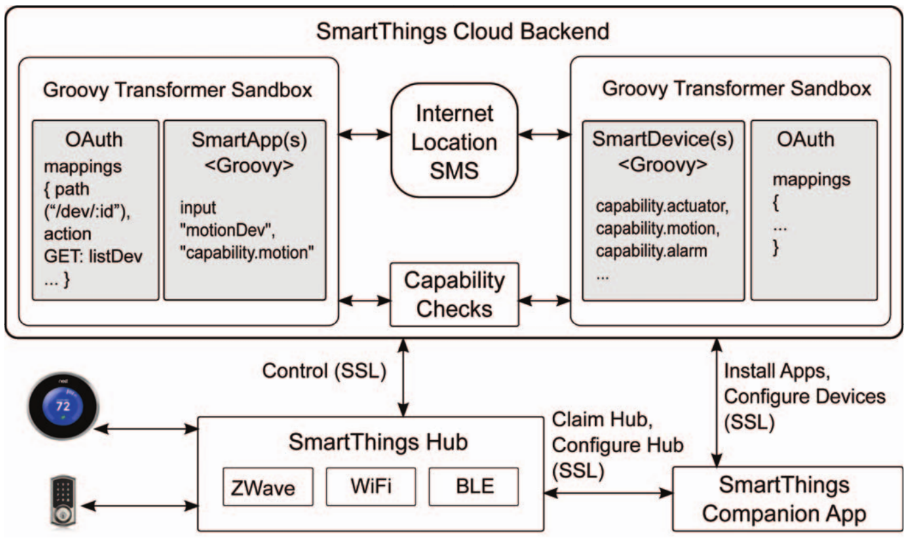

Figure 1: The complete SmartThings framework

Each SmartThings user needs to have a hub. This hub communicates with the physical devices in the home via radio protocols. The hub gets controlled by a smartphone companion app. This companion app gives access to a number of custom made SmartApps and SmartDevices. SmartDevices are software wrappers used to control the physical devices. SmartApps communicate with SmartDevices in order to achieve certain goals like locking the physical door. Although SmartApps and SmartDevices are managed by the companion app, they are *executed* in the closed-source cloud back end of Samsung. All of the above components of the SmartThings framework communicate via a SSL-protected connection. This makes it more difficult to analyze the specifics of the framework.

SmartThings enables SmartApps to send information via SMS or the Internet. The API is unrestricted so malicious SmartApps can leak any obtained sensitive information this way.

2.1.2 Events

SmartThings uses an event subsystem. When SmartDevices want to asynchronously communicate information to SmartApps they can issue events containing the information. Authorized SmartApps can *subscribe* to these events and consequently can receive the messages.

The event-driven processing in the SmartThings-framework is not secure. There are multiple ways by which SmartApps can snoop on events even without authorization. This can lead to SmartApps unrightfully accessing SmartDoor lock pin-codes. Section 3.2 expounds the ways event leakage can happen.

Additionally, there is a way in which events can be spoofed. This is further explained in Section 3.2.3 and an attack demonstration is discussed in Section 3.4.4.

2.1.3 Third-Party Communication by WebService Integration

SmartApps are able to communicate with third-party apps if this feature is integrated by the developer of the SmartApp. If the functionality is integrated then the SmartApp makes Web service endpoints available which can be talked to via HTTP requests. The third-party app may make a request to the SmartApp which leads to the SmartApp executing the specific code block the developer intended for the request.

The SmartApp with WebServices possesses a client ID and a client secret (both of which are 128-bit random values). In order for the third-party app to make a valid request the app needs to be verified. It needs to know the SmartApp client ID and secret and also has to redirect the user to an official SmartThings website where they must log in with their credentials. If the third-party app fulfills these conditions then it gains a valid OAuth token to authenticate with the SmartApp it wants to access. The vulnerabilities of this authentication method are illustrated in Section 3.3.1.

2.2 Survey Results

Earlence Fernandes et al. [1] conducted a survey on different user groups owning a Samsung's SmartThings Framework at home with at least 10 SmartApps installed. Most of the researchers in Earlence Fernandes et al. [1] team did not access the survey research at all. Instead, they assigned a handful of their members from other institutions to conduct the survey. It was pointed out, that the users data (e.g. name, address etc.) was not fully collected. Only the emails were, so that the users, who willingly agreed to participate for a reward, could receive their gift cards. Earlence Fernandes et al. [1] did not tell those users that their answers were supposed to be used as a security study, in order to not make them think differently about the survey. Instead, they mentioned their study as a *SmartThings app installation experience.*

The point of the survey was to examine, e.g. determine what a user would or would not install as a SmartApp. The survey consisted of several questions divided into sections.

Users in the first section were asked to give their opinions on a battery monitoring SmartApp. That SmartApp could access several other devices that were connected to their hub. The users had to answer whether they would like this app, later described in Section 3.4.2, to access their sensitive data (e.g. pin-codes) or not. Furthermore, they were asked what devices they would like to be monitored by the battery monitoring SmartApp. The devices connected were a motion and presence sensor, a door lock and a siren strobe alarm.

Users in the second section were asked how likely they would install that SmartApp given the previous information.

The following section regarded their knowledge on security, regarding the installation process and the risks that come with it. Users had to answer questions about what actions could be performed by the battery monitoring SmartApp besides the battery monitoring. Those extra actions included *Send out the SmartThings motion and presence sensor's events to a remote server*, *Collect door access codes in the Schlage door lock and send them out to a remote server* and more. In Section 3.4.2 we explained how the battery monitoring SmartApp works. Earlence Fernandes et al. [1] noted that this SmartApp only needed access to the devices in order to send data to a remote server.

The last section regarded the users and their anger if they were to found out that their security had been compromised.

Below is a Table with the collected data from the survey done by Earlence Fernandes et al. [1].

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

Table 1: Survey results

3 Methodology

3.1 Capabilities: The Broken Privilege System

The SmartApps are written in a restricted subset of the Groovy-language. In order for a SmartApp to control and/or communicate with a physical device it needs to connect with the corresponding SmartDevice. To act on the device the SmartApp needs to request the permissions to do so. In the SmartThings framework these permissions are called *capabilities*.

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

Table 2: Example SmartDoor capabilities with their commands and attributes

Table 2 contains example capabilities of a SmartDoor. E.g. the lock capability gives the SmartApp access to the *lock()* and *unlock()* methods. The associated attribute - *lock* - is the lock status of the SmartDoor.

Earlence Fernandes et al. [1] found out that in many cases the SmartApps gain more permissions than they explicitly ask for. This is due to the design of the SmartThings framework. A certain amount of SmartApps even use unrequested capabilities and would no longer function if these capabilities were retro-actively revoked. That makes patching the SmartThings framework difficult. There are two ways in which SmartApps can become overprivileged.

3.1.1 Coarse-Grained Capabilities

When SmartApps request explicit capabilities then in many cases they also get multiple associated capabilities. An example: When an app needs the ability to lock a door and requests the *lock* capability. This *lock* capability however also gives the app the capability to unlock the door. This circumstance cannot be evaded, the SmartApp developers have no choice in this.

3.1.2 Coarse SmartApp-SmartDevice Binding

As clarified in Section 2.1, each SmartApp needs to communicate with a SmartDevice in order to access the correlated physical device. When the SmartApp user opens the app for the first time, they will be prompted to authorize a SmartDevice to be used. The problem is that SmartThings puts no value on how many capabilities the SmartDevice grants the app as long as it grants at least all the capabilities which the app requests. The result is that SmartApps automatically gain way more capabilities than they need for their functionality. The app developers have no choice in this circumstance as it is the framework itself which leads to this overprivilege.

This amount of overprivilege can be illustrated with the standard ZWave lock SmartDevice which is used to control the physical SmartDoor. Among others the SmartDevice gives access to the capabilities *capability.lock*, *capability.lockCodes* and *capability.battery.* If now a SmartApp *only* wants to monitor the battery status of a ZWave door and the user authorizes the app with the standard SmartDevice depicted above, then that app would also have the capabilities of locking/unlocking the door and also setting the lock codes. A more specific example is given in Section 3.4.2.

3.2 Event Leakage and Event Spoofing

3.2.1 SmartApp-SmartDevice Binding Leads to Event Leakage

As soon as a SmartApp is allowed to bind to a SmartDevice, it then automatically gets the ability to subscribe to all of the events the SmartDevice publishes. There is no separate capability which needs to get requested in order to receive events. Events are explained in Section 2.1.2.

This circumstance is a critical security vulnerability as multiple SmartDevices broadcast sensitive information with events. An example is the ZWave lock SmartDevice. It communicates lock-pin codes in *codeReport* events. A SmartApp could log the lock-pin codes to a SmartDoor even if it only needs trivial access to the SmartDevice, e.g. for measuring the battery status. An example for this attack is given in Section 3.4.2.

3.2.2 Device Identifier Implementation Enables Eavesdropping

Each SmartDevice is identified by a unique 128-bit device identifier. This identifier is random and gets assigned once the SmartDevice first gets paired with the hub. SmartApps can register for receiving events of a SmartDevice by subscribing to the SmartDevice. The 128-bit device identifier is used for this. Even if a SmartApp is not bound to a SmartDevice, it can still receive the event information of the device if it knows the device identifier. SmartApps knowing the device identifier of a SmartDevice therefore can eavesdrop on all the data the SmartDevice communicates via events.

It is easy for a SmartApp to get to know the device identifier of a SmartDevice. SmartApps can communicate with each other. SmartApps which do have validation to know the device identifier of a SmartDevice can transmit the identifier to a malicious SmartApp via the OAuth protocol. The malicious SmartApp does not need any authorization for this attack whatsoever.

3.2.3 Spoofed Events

There is no access control for raising events and SmartApps also have no way of checking whether an event they receive is legitimate or malicious. Attackers are able to easily spoof physical device events and also location-related events.

An example for a physical device event is a smoke detector SmartDevice which transmits the information when the physical device detects smoke. A malicious SmartApp can propagate false events to all subscribed SmartApps just as if a SmartDevice would have send it. All the attacker needs are the hub ID, location ID and the device identifier. The first two are available to all SmartApps by default. The device identifier can easily be obtained. This is explained in Section 3.2.2.

In the SmartThings framework there is also a *location* object which all SmartApps can access. SmartApps can depend on it, e.g. an app which enables vacation mode. Attackers are able to spoof location events and can even disable the vacation mode of the SmartHome. This insecurity can lead to burglary. An example attack is outlined in Section 3.4.3.

3.3 Other Security Vulnerabilities

3.3.1 OAuth Insecurities

As described in Section 2.1.3 an OAuth token is needed in order for a third-party app to request features of a SmartApp. This authentication method, however is fundamentally flawed as the protocol frequently gets implemented incorrectly by app developers. This is among other reasons due to developers misunderstanding the protocol and also the poor documentation of it.

On multiple occasions developers of third-party apps embedded the client ID and secret of a SmartApp into the bytecode of their third-party app. The result of this is that anyone who extracts the ID and secret out of the bytecode is now able to obtain a fake, but valid OAuth token. This token can be used to gain access to the corresponding SmartApp. A demonstration of this will follow in Section 3.4.1.

3.4 Attack Demonstrations

Their study is based on four security design flaws which are exploited using WebServices, command injection and snooping. The Table below depicts the attacks Earlence Fernandes et al. [1] demonstrated in their paper.

Attack Description	**Attack Vectors**	**Physical World Impact (Denning Classification)**
Backdoor Pin Code Injection Attack	Command injection to an existing WebService SmartApp; Overprivilege using SmartApp-SmartDevice coarse-binding; Stealing an OAuth token using the hard-coded secret in the existing binary; Getting a victim to click on a link pointing to the SmartThings Web site	Enabling physical entry; Physical theft
Door Lock Pin Code Snooping Attack	Stealthy attack app that only requests the capability to monitor battery levels of connected devices and getting a victim to install the attack app; Eavesdropping of events data; Overprivilege using SmartApp-SmartDevice coarse-binding; Leaking sensitive data using unrestricted SMS services	Enabling physical entry; Physical theft
Disabling Vacation Mode Attack	Attack app with no specific capabilities; Getting a victim to install the attack app; Misusing logic of a benign SmartApp; Event spoofing	Physical theft; Vandalism
Fake Alarm Attack	Attack app with no specific capabilities; Getting a victim to install the attack app; Spoofing physical device Events; Controlling devices without gaining appropriate capability; Misusing logic of benign SmartApp	Misinformation; Annoyance

Table 3: Four proof-of-concept attacks on SmartThings

3.4.1 Command Injection Attack

The first proof-of-concept attack is done with command injection. Command injection means remotely sending a string e.g. command over a protocol (OAuth) using method invocation, overprivileged methods and insecure third-party integration. Earlence Fernandes et al. [1] used a popular third-party Android app for their demonstration. Its function was to make interactions and management of SmartDevices easier. The first part of their command-injection attack was to acquire a (OAuth) token. They had to get the user to use a link to a SmartThings domain, which was modified with the attackers' domain, in order to obtain that token. The user had to log into his device, afterwards which he was redirected to the attackers' domain. With that redirection, the attacker gained the users' client ID and client Secret trough the URI. This data is needed to complete the OAuth token. The only thing left for the attacker to do was to combine that information with his. SmartThings allows access to the corresponding SmartDevice to anyone that has such a token. That obtained OAuth token is needed for the second part of the command injection attack. The third-party Android app made use of that token and reconstructed the format of command strings used by the WebService SmartApp. Earlence Fernandes et al. [1] tested what commands are accepted by the WebService SmartApp and determined that all commands were. Part of the reason behind this was that the WebService SmartApp did not check the input of its used methods. The methods did not have any sanitization of the yet to be used commands whatsoever. But they did not need all commands, but rather just one - *setCode*. The next step was to send new lock codes with that command over to that SmartApp. After that they just had to verify if the backdoor pin-code was indeed modified - and it was. The command used was a member of a capability of SmartApps (capability.lockCodes) which was automatically acquired through the design of SmartThings. At this point, the attacker did not need the users hub to be online in order for the attacker to unlock the door. He could enter the house whenever he wanted to. Below is a portion of a *Logitech Harmony WebService SmartApp* used in Earlence Fernandes et al. [1] where lines 19 and 20 show exactly why this could happen.

```
mappings {
  path("/devices") { action: [ GET: "listDevices"] }
  path("/devices/:id") { action: [ GET: "getDevice", PUT: "updateDevice"] }
  // --additional mappings truncated--
}

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

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

Figure 2: Unsafe use of Groovy dynamic method invocation

3.4.2 Door Lock Pin Code Snooping Attack

The second proof-of-concept attack is stealthy and done with a battery monitoring SmartApp specifically designed with malicious intent. From the users' perspective, it is just monitoring the battery status of SmartDevices paired with SmartThings, but what the user does not know is that it allows the attacker to break into the home. Earlence Fernandes et al. [1] use a Schlage lock FE599 which operates using a SmartApp. While installing the SmartApp, the user is asked for things like battery permissions. The user is then asked for a new pin-code after the installation has finished. After completion, the pin-code is being sent as a signal through a protocol named ZWave over to the hub. If the device handler of the SmartApp receives an acknowledgment from the hub, that the signal was received, it creates an event object containing that plain text pin-code. The battery monitoring SmartApp can continue snooping for all types of such event objects. From that point forward it is easy to understand that the malicious SmartApp can leak this data to the attacker. The picture below from Earlence Fernandes et al. [1] visualizes the attack steps.

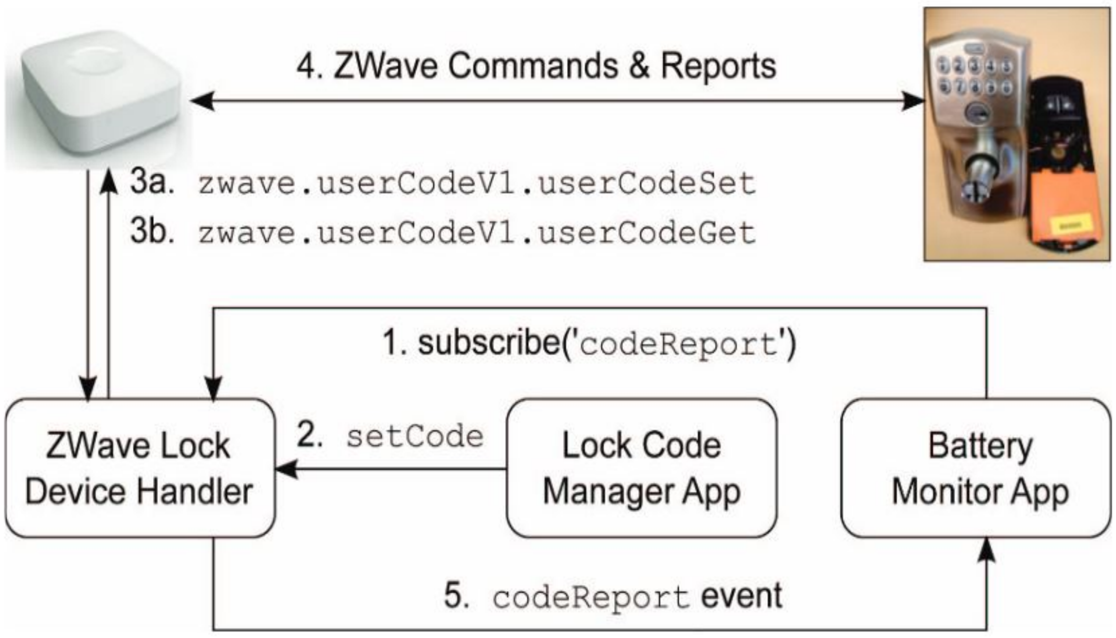
4. ZWave Commands & Reports
3a. zwave.userCodeV1.userCodeSet
3b. zwave.userCodeV1.userCodeGet
1. subscribe('codeReport')
ZWave Lock Device Handler
2. setCode
Lock Code Manager App
Battery Monitor App
5. codeReport event

Figure 3: Ordered snooping attack steps

Although the battery monitoring SmartApp appeared to be asking for just battery capabilities, it gained authorization to other capabilities as well. That is overprivilege due to coarse bindings. The fact that the device handler of SmartThings leaks plain text data (e.g. pin-codes) to other authorized devices, is another fundamental issue here. Earlence Fernandes et al. [1] took the leaking a bit further and enhanced the snooping attack with event leakage. Earlier in Section 3.1.2 we explained that once the 128-bit device ID is known to our SmartApp, it can be used to listen to all events from that device. That device ID is constant amongst all SmartApps connected to SmartThings, but changes once a device is removed. So, there has to be a way to get it. There are indeed a few ways to acquire that device ID, one of which is using a WebService like earlier. Once that ID is leaked, the attacker can transmit it to some malicious SmartApp. That SmartApp can use the leaked device ID to disguise itself as the original SmartDevice. Hence, it can be used for malicious intents.

3.4.3 Disable Vacation Mode Attack

The third proof-of-concept attack is using a connection between a property and event to disable vacation mode. The SmartApp used for this requires a location object to distinguish whether the user is at home or not. It then turns on or off the lights depending on the mode property of that object. This is to prevent burglars breaking into the users' home. SmartThings however does not have security controls for that connection. Earlence Fernandes et al. [1] used the intended SmartApp to misuse this design flaw and trigger a mode change event. This disabled the protection associated with vacation mode and it *required only one line of attack code*[1].

3.4.4 Spoofed Events Attack

The fourth proof-of-concept attack is using spoofed physical device events. The downloaded App could monitor and trigger events of (CO) detectors (e.g. fire alarm systems, motion sensors and etc.). Earlence Fernandes et al. [1] made use of a spoofed physical device event to trigger such detectors and with that misuse the logic of such systems.

4 Evaluation

There are multiple key takeaways from the study. The most important one was that the underlying programming framework is the one most central piece when it comes to the security of SmartHome technology. SmartApps, physical devices and communication protocols could only be as secure as the framework allows.

In the most widespread framework at the time the study was carried out, the SmartThings framework, the capability model was especially crucial in making the SmartHome technology vulnerable. Overprivilege was possible due to coarse-grained capabilities and also coarse SmartApp-SmartDevice binding (see Section 3.1).

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

Table 4: Distribution of reasons for overprivilege

Also harmful for the security of the SmartThings framework was the lack of sane security standards in the implementation of the SmartApps and SmartDevices themselves. Command injection attacks made possible by the Groovy implementation and also failed OAuth realization are key points here. The event system allows malicious SmartApps to snoop on events and also to create illegitimate events.

The total number of vulnerable SmartApps and SmartDevices can be observed in Table 5.

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

Table 5: Number of vulnerable SmartApps and SmartDevices

Multiple proof-of-concept attacks could be carried out by Earlence Fernandes et al. [1] (Section 3.4) which prove the legitimacy of the flaws found in the SmartThings framework.

Finally, the survey carried out by the researchers shows that even SmartHome-hobbyists were generally not aware of the dangers of the SmartThings framework. Because of this it is even more unlikely that people of the mainstream are aware of how dangerous the Smart technology can be they are using.

5 Related Work

5.1 Fingerprinting Mainstream IoT Platforms Using Traffic Analysis

The paper by Yang et al. [3] presents the reality of modern networks and dives into the vulnerabilities of network traffic. Their research makes use of the connection between the broad use of IoT devices and cloud servers as well as Smart home applications. It is based on leading mainstream IoT cloud platforms such as Samsung's SmartThings, Samsung Connect, Alibaba Alink and others. The reason why Samsung SmartThings was picked was, because back then it had a lot in common with the mainstream IoT model. That model consists of a companion mobile app (SmartApp), a cloud server (SmartThings) and a device (SmartDevice). The publishers first came up with a new software and mechanism that helps distinguish traffic between cloud servers and mobile communication. That mechanism is called fingerprinting - the technique of evaluating operating systems by extracting data of outgoing internet packets and distinguishing that data from known operating systems. Attackers can use that information to select an appropriate attack mechanism for that specific operating system. They pointed out that fingerprinting in the world of IoT is similar to that of the operating system. In their opinion, the research of fingerprinting in the world of IoT is helpful for servers and large-scale network traffic to prevent such attacks.

They conducted their research with identified traffic and found out, that the sensitive information sent by IoT platforms did not only contain device control commands and device information, but also non-IoT traffic. This is similar to Samsung's SmartThings platform where the bad design allowed SmartApps not only to use their own capabilities, but also those of others. Furthermore, spoofed SmartApps were able to collect sensitive data and transmit it back to the attacker.

Part of the research was done on traffic decryption of the hypertext transfer protocol with the man-in-the-middle attack. For that they used a tool named Fiddler. The main challenge with that tool was to install a root certificate, provided by the tool, inside the target device. That certificate would allow all traffic to be forwarded through the man-in-the-middle device over to them.

They pointed out that the research of [1] served as an introduction to the vulnerabilities of new IoT platforms. That in mind, and other related research on this topic, helped them develop a traffic marking ("fingerprinting") tool named IoTPF. Their fingerprinting tool firstly needed to decide whether the received packets were of the HTTP or HTTPS protocol, or not. The tool had to abort any operations if the protocols were not HTTP or HTTPS. After this condition succeeded, the tool had to split up the response and request of the protocol into two different modules which information were needed for the marking results. The vital information, needed for the identification of IoT business data, was wrapped inside the HTTP header. Samsung's SmartThings as well as Amazons AWS use that same protocol. The data inside the header was IoT and non-IoT specific and Yang et al. [3] only focused specifically on the IoT part. With that being said, their research was not based on devices specific functionality, but on their basic information inside the used protocols.

Yang et al. [3] pointed out that fingerprinting operating systems of remote servers can be used for penetration testing. IoT Platform Markets are marked by their platform, which can uncover internal relationships. Intrusion detection systems can be evaluated by using a man-in-the-middle attack which can uncover vulnerabilities inside the system.

5.2 Lightweight and Privacy-Preserving Remote User Authentication for Smart Homes

The paper by Joong-Lyul Lee et al. [2] focuses on the security of IoT and generally embedded devices. In particular they are interested in keeping SmartHome technology safe and secure. Instead of dealing with the framework like Earlence Fernandes et al. [1], Joong-Lyul Lee et al. [2] are involved in the resilience of the communication protocols used within the framework. Just as we described in Section 3.3.1, they warn about currently implemented authentication protocols like OAuth authentication. In particular, they point out the rise of phishing attacks and attempts to steal SmartHome users' credentials. According to them, the increased amount of remote work and remote schooling resulting from the COVID-19 pandemic have led to a great rise in these kinds of credential stealing attacks.

In order to show examples of IoT security vulnerabilities Joong-Lyul Lee et al. [2] cite multiple papers including *Security Analysis of Emerging Smart Home Applications* [1].

However, the focus here is more on the communication protocols used in Smart Homes, for example *ZigBee* and *ZWave*.

As a solution to these protocol insecurities they present an alternative, lightweight protocol which can realize user authentication. The central advantage of this protocol is the drastic reduction of computational overhead. Comparisons to other protocol schemes reveal that their protocol has a 20% reduction in communication overhead on Smart Devices.

The efficiency of this protocol simplifies the communication making it more resilient against phishing attacks. Additionally it saves resources which is especially important in embedded systems and IoT devices.

Geometric secret sharing is another feature of the protocol. Multiple devices can mutually authenticate each other. Users of IoT devices no longer need to remember passwords and smart cards become obsolete.

The protocol gets tested on a Raspberry Pi which enables the measurement of the power consumption.

6 Possible Measures to Prevent Future Security Risks in Smart Home Technology

The design model has to be used differently. Operations are to be divided in order to prevent hazards e.g. turning alarms and ovens on and off frequently. It makes sense for operations to be divided based on their provided functionality and risks. Earlence Fernandes et al. [1] suggest that it would be of great use for the separation of operations *to include input from multiple stakeholders: users, device manufacturers, and the framework provider.* Granularity increases with more separation. However, this increasing comes with a price - difficult use of the system by the user. Although some home systems try to avoid risk-based granularities with different techniques, the lack of standard persists.

As demonstrated in Section 3.4 two problems persist - event identification and sensitive event data handling. Their studies revealed that an app can monitor events once it knows the ID of a smart device. One can easily understand why a more secure design is needed. One approach of solving this design flaw could be the use of an mechanism that verifies the origin of an event and separate sensitive from crucial data. They suggest that this could be *the basis for a more secure event architecture.* Such a secure event architecture can be seen in Android Intents where each Intent is built on unique IDs. Those unique IDs are created by the kernel. That way a receiver can check ownership of information provided to him. As for the sender - he can decide whether to send out information at all or not. This would not work without unique IDs.

Earlence Fernandes et al. [1] stress that modern techniques against code injection attacks and overprivileged SmartApps are needed. So far the issues remained in the hands of the framework designers e.g. Section 3.1 and Section 3.4. But there is one place where final validation e.g. vetting takes place - the App Stores used to provide this kind of software. App Stores have to start validating SmartApps more securely, but it is unlikely for them to discover such vulnerabilities on their own. This could be achieved if Stores and framework providers cooperate. That way validation of secure SmartApps could be improved and security vulnerabilities removed.

7 Conclusion

The concerns of privacy-invasion and potential theft have been raised, but not yet fully resolved. Earlence Fernandes et al. [1] succeeded in exploiting most of these vulnerabilities explained in Section 3. This valuable information have been forwarded to Samsung, as well as to all other related companies and institutions. The achievement of Earlence Fernandes et al. [1] helped companies to redesign their future frameworks and update their already existing ones.

The question remaining is how much of these vulnerabilities can be fixed. Earlence Fernandes et al. [1] gave an idea of what companies should consider doing. They should cooperate and collaborate with other companies instead of designing software alone. This should improve the functionality and security of SmartHome Devices and others. It is not only the responsibility of companies to ensure security. Earlence Fernandes et al. [1] did also suggest that users need to keep their devices firmware up-to-date in order to prevent privacy theft and others.

There have been made improvements in this area already. Samsung acknowledged their design flaws and worked on their products ever since. Most of the here described vulnerabilities have been resolved (e.g. the insecure ZWave protocol has been improved). But one must not forget that there is no such thing as perfect software and thus there will always be room for improvement.

References

[1] Earlence Fernandes et al. *Security Analysis of Emerging Smart Home Applications.* IEEE, 2016.

[2] Joong-Lyul Lee et al. *Lightweight and Privacy-Preserving Remote User Authentication for Smart Homes.* IEEE, 2022.

[3] Yiyu Yang et al. *Fingerprinting Mainstream IoT Platforms Using Traffic Analysis.* IEEE, 2022.