Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 7849065, 15 pages
https://doi.org/10.1155/2018/7849065

Research Article

Resetting Your Password Is Vulnerable: A Security Study of Common SMS-Based Authentication in IoT Device

Dong Wang[1], **Xiaosong Zhang**[1], **Jiang Ming**[2], **Ting Chen**[1], **Chao Wang**[3] and **Weina Niu**[1,4]

[1] *University of Electronic Science and Technology of China, China*
[2] *The University of Texas at Arlington, USA*
[3] *ADLab of Venustech, China*
[4] *College of Cybersecurity, Sichuan University, China*

Correspondence should be addressed to Xiaosong Zhang; johnsonzxs@uestc.edu.cn

Received 8 March 2018; Revised 28 May 2018; Accepted 4 June 2018; Published 4 July 2018

Academic Editor: Ximeng Liu

Firmware vulnerability is an important target for IoT attacks, but it is challenging, because firmware may be publicly unavailable or encrypted with an unknown key. We present in this paper an attack on Short Message Service (SMS for short) authentication code which aims at gaining the control of IoT devices without firmware analysis. The key idea is based on the observation that IoT device usually has an official application (app for short) used to control itself. Customer needs to register an account before using this app, phone numbers are usually suggested to be the account name, and most of these apps have a common feature, called *Reset Your Password*, that uses an SMS authentication code sent to customer phone to authenticate the customer when he forgot his password. We found that an attacker can perform brute-force attack on this SMS authentication code automatically by overcoming several challenges, then he can steal the account to gain the control of IoT devices. In our research, we have implemented a prototype tool, called *SACIntruder*, to enable performing such brute-force attack test on IoT devices automatically. We evaluated it and successfully found 12 zero-day vulnerabilities including smart lock, sharing car, smart watch, smart router, etc. We also discussed how to prevent this attack.

1. Introduction

The Internet of Things (IoT for short) paradigm is one of the most thrilling innovations of the recent years. Growing interest has spurred the commoditization of many devices for personal use, such as smart home devices, smart wearable devices, and smart car [1, 2]. The figure of online capable devices increased 31% from 2016 to 8.4 billion in 2017. Experts estimate that the IoT will consist of about 30 billion objects by 2020. It is also estimated that the global market value of IoT will reach $7.1 trillion by 2020 [3].

For an IoT device, it usually consists of three components, an electronically augmented hardware device that reports its status and processes user commands, a mobile device that is used to receive status and send commands, a cloud that is used to exchange messages between the hardware device and mobile device. When a customer deploys his device, he will (1) install the device, (2) download the official app and install it on his smart phone, (3) register an account via the app, (4) pair the device with his smart phone via the app, (5) start to control his device via the app. Usually, the device can be controlled remotely via the app. So, if there are vulnerabilities in the device, an attacker can also control it remotely. While prior works on IoT focused on cryptographic protocols analysis [4–6], limited work has studied on security vulnerabilities of implementation.

In this paper, we study the common SMS-based authentication that is used in IoT devices. We observed that many IoT devices support a common feature, called *Reset Your Password*, designed for a customer to change his account password if he forgets it. To authenticate the customer, IoT cloud will send an authentication code to customer phone that is registered as the account name via SMS. These SMS messages share the same structure, *[sender] [text with authentication code] [expiration]* (e.g., *Panasonic: Your verification*

code is 3895, validity period is 5 minutes). SMS authentication code is usually a 4-digital or 6-digital number, here is *3895*. Ideally, the code is just known by the IoT cloud and the device owner, so anyone who can present the code will be considered as the real device owner. After analyzing several IoT devices, we found this SMS-based authentication of *Reset Your Password* may be vulnerable. An attacker can perform brute-force attack via mutating the SMS code in password reset message and reset IoT account password, because the search space of the digital SMS code is very small.

However, there is a big challenge: *cryptographically consistent message* (Section 3). If we modify the SMS code in a password reset message, it will become *cryptographically inconsistent* because the message may contain a signature, and this will cause the message being discarded by IoT cloud. We found that we can treat IoT app as a black box and control its execution, then we can reuse the app code to generate cryptographically consistent messages when we mutate the SMS code. We also addressed several other challenges that prevent us performing the attack test automatically. We proposed the design of our prototype tool (Section 4) *SACIntruder*; it can be used to check whether a IoT device is vulnerable to the SMS-based authentication automatically. We implemented our tool and used it to find 12 zero-day vulnerabilities. For instance, we found a vulnerable smart lock and an attacker can enter into a victim's house without authorization. We also found a vulnerable car and an attacker can drive it away.

In short, we make the following major contributions:

(i) To the best of our knowledge, it is the *first* security study about the SMS-based authentication in IoT device, and we found it may be vulnerable. An attacker can perform brute-force test on SMS code to gain the control of IoT devices without any interaction of victims.

(ii) We designed a tool *SACIntruder* that can automatically perform brute-force attack test to check whether an IoT device is vulnerable to SMS code. Our tool addressed the big challenge about cryptographically consistent message generation and other several challenges such as UI identifying, parameter identifying, and time expiration.

(iii) We implemented our tool and evaluated it on IoT devices including smart lock, sharing car, smart watch, and smart router, and it found 12 zero-day vulnerabilities automatically. We already reported all of them to the *CNCERT/CC* [7] to help vendors to fix them, and they all have been fixed now.

The remainder of this article is structured as follows. In Section 2, we introduce the background knowledge of IoT security. Then, we use a home app as an example case study to understand our problem and present the overview of *SACIntruder* in Section 3. We present the detailed design in Section 4 and evaluate it in Section 5. We discuss how to prevent this attack with the goal of security and usability in Section 6. The survey of related work in Section 7 is followed by our conclusion in Section 8.

2. Background

Traditional embedded devices are offline, and they can be controlled just physically. In contrast, many IoT devices are online and can be accessed via the Internet. So, an attacker can gain the control of these devices remotely, if there are security flaws. The loose protection and pervasiveness of vulnerabilities [8, 9] make these devices very weak to attackers. For instance, there are more than 90 reports about independent IoT attack incidents from 2014 to 2016 [10].

For IoT attacks, firmware is always an important target, because security vulnerability in firmware usually can bypass all limitations such as the accessibility of the underlying system, and an attacker can find a large number of vulnerabilities by analyzing firmware because it contains all critical codes. In 2017, F-Secure security researcher analyzed the firmware of a Foscam IP camera and found 18 zero-day vulnerabilities including insecure default credentials, command injection, stack-based buffer overflow, etc. There are some works about detecting vulnerabilities in firmware, some utilize symbolic execution [11, 12] to detect flaws automatically, while others construct an emulation runtime for dynamic analysis [13–15]. However, firmware acquisition is a big challenge for detecting vulnerabilities via firmware analysis, because not every device firmware is publicly available. Even if available, it may be encrypted with an unknown cryptographic algorithm or data key. In addition, firmware is usually a compressed archived file, it is unable to decompress it without the knowledge of the archived file format. The diverse architectures of hardware chipset is another challenge for firmware analysis, because different chipset has different memory layout and instruction set.

Many IoT devices can be controlled by a customer through the official apps with his IoT account (e.g., a smart lock that enables its owner to open or close the door remotely). So, if an attacker can compromise the account, he can gain the control of the device via the app. There are some methods for cracking an account. Password Brute-force attack [16] is a traditional account attack by trying all possible passwords. After analyzing a lot of IoT apps, we observed that most IoT accounts have the same password strategy: at least 6 characters and each character can be *A-Z*, *a-z*, and *0-9*. So, the size of password search space will be $(26 + 26 + 10)^6 = 56$ billion, it will take quite a long time to test every possible password. Cross-Site Scripting [17] (XSS for short) and Cross-Site Request Forgery (XSRF for short) [18] can also be used to take over an account, but they are special for browser and most IoT apps are not built on a browser. So, XSS and XSRF rarely impact IoT account. Phishing [19] is another important method to steal an account, an attacker runs an evil website that is very similar to the target website and guides a victim to access it and input his credential. However, IoT customers are typically guided to install apps from the vendor cloud or the official app store by the device manual. Therefore, it is challenging for a phishing attacker to inject the download of IoT apps to get the victim credential. Man in the Middle [20] (MitM for short) attack is another common way to steal a user account. The recent work, Password Reset MitM [21] (PRMitM for short) attack, exploits the similarity

```
GET /ci/user/getVerifyCode?uid=-1&phone=1383815****
&imei= HTTP/1.1
Host: * * *.* * *house.com.cn
Connection: Keep-Alive
Accept-Encoding: gzip
User-Agent: okhttp/3.3.1
```

Box 1: Messages of the password reset for a home app. App Request for a SMS authentication code.

```
{"code":200,"msg":"","result":{"session_id":"f7b532
83-3c20-400d-b0ee-76a171036414","code_-1":-
1,"error_code":"0"}}
```

Box 2: Messages of the password reset for a home app. Cloud Response for the SMS authentication code.

```
POST /ci/user/fgt/pwd?password=e10adc3949ba59abbe56
e057f20F883e&code=7496&phone=1383815****&sign=d3db1
a89d68cd72cbd2 a3fcbf9822876 HTTP/1.1
Cookie: JSESSIONID=f7b53283-3c20-400d-b0ee-76a17103
6414
Content-Length: 0
Host: * * *.* * *house.com.cn
Connection: Keep-Alive
Accept-Encoding: gzip
User-Agent: okhttp/3.3.1
```

Box 3: Messages of the password reset for a home app. App Request for password reset.

```
{"code":0,"msg":"reset success"}
```

Box 4: Messages of the password reset for a home app. Cloud Response for password reset.

of the registration and password reset processes to launch a MitM attack to popular websites and mobile apps. However, PRMitM relies on victim's interaction heavily to retrieve everything that is essential for password reset. So, the success of PRMitM attack relies on several strong assumptions. First, it requires that the victim registers an account or inputs his mobile phone and SMS authentication code to the compromised website. Second, it assumes that many victims will ignore the details of the password reset message but just copy the SMS code into the compromised website. Third, an attacker has to analyze target website to get the knowledge of every challenge. The attack will not happen if any of the above assumptions does not meet.

As stated early, the search space of the SMS code used in *Reset Your Password* is very small and we can perform brute-force attack on it. Once our mutation meets the right value before the time expiration, we can reset the password of IoT device account. In contrast to our previous work [22], we have presented a new UI model, a new approach to identify parameters and an approach to reduce unnecessary requests. So, we can find new vulnerabilities that cannot be found via our previous work.

3. Overview

The goal of this work is to understand the SMS authentication of *Reset Your Password* in IoT devices and automatically identify whether a device is vulnerable. In this paper, we focus on IoT apps in Android platform that is the most popular mobile platform in the world [23]. We first use a running example to discuss our problem in Section 3.1; we discuss the major challenge in Section 3.2 and other implementation challenges in Section 3.3 and then give an overview of *SACIntruder* in Section 3.4.

3.1. A Running Example of Password Reset. To understand our problem better, we select an android IoT app that is designed to control smart home devices. Boxes 1–4 illustrate

the messages used for the whole progress of the password reset.

When a customer wants to reset his IoT account password, he will be guided by app UI to send a message to the cloud for an SMS code, then the cloud will generate a code, send it to customer phone, and respond with a message to the app. Boxes 1 and 2 present a pair of messages about request and response. In theory, there is no limitation for the message format between the app and cloud. After analyzing a lot of apps, we found that most of them are built on Hypertext Transfer Protocol (HTTP for short) [24]. One possible reason may be that Representational state transfer (REST for short) [25] is popular in the development domain, and REST uses HTTP as its low-layer transport protocol. The Uniform Resource Locator (URL for short) [26] of REST usually has full meaning. In the running example, *getVerifyCode* and *phone=1383815∗∗∗∗* mean requesting the cloud to send an authentication code to *1383815∗∗∗∗*.

After receiving the SMS authentication code, customer inputs the code and new password and clicks a UI component to send a message containing the code and password to the cloud, then the cloud verifies them to replace the old password, at last responds with a message to the app. Box 3 presents a password reset request message, it contains four key parameters, three of them (*password, code, phone*) are mapped to user inputs, while *sign* cannot be mapped to any input. Moreover, *sign* is usually generated by a cryptographic algorithm with arguments including *password, code, phone* and other app-specific data. It is easy to modify the *code*, but *sign* will prevent us from doing this. Because if we modify it, we must update *sign*, or the message will be cryptographically inconsistent and the cloud will discard it. Besides the cryptographically consistent message generation, several implementation challenges also need to be addressed, we discuss them in the following sections. Box 4 presents a password reset response message for successful password reset.

In addition, some IoT apps only support to login the account with SMS authentication code, the customer never owns a password. He will request an SMS code every time when he wants to login the account. This type of login can be classified as a special password reset that contains no new password. So, we can support these apps using the same brute-force attack method.

3.2. Cryptographically Consistent Message Generation. The SMS code in *Reset Your Password* is a digital number with small search space, it is easy for the brute-force attack. But many IoT apps use at least one cryptographic strategy to protect messages, some chose signature to ensure the integerity [27], while others chose encryption to ensure the confidentiality [28]. IoT cloud will check the confidential or integrity of a message. If not valid, discard the message. We must find an approach to generate cryptographically consistent messages while mutating the code.

A straightforward method is to extract the cryptographic algorithm from the app and then reimplement it in the outside of IoT apps. Some program analysis technologies such as symbolic execution [29–31] and taint analysis [32, 33] can be applied to do this. These technologies can analyze a program to determine what inputs cause each part of the program by monitoring the execution of every instruction and its referenced data, so they are widely studied to explore the internal status of a program. They work well for many program logic, except the cryptographic algorithm because of the well-known challenge named *path explosion* [34, 35]: the number of feasible logic paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations. Unfortunately, loop iterations are very common in the cryptographic algorithm. IoT apps are commercial software, they usually contain complex logic and developers prefer to deploy some protections to enhance the security. In addition, packer is widely used to protect an app by developers, it contains code obfuscation [36], resource encryption, antidebugging, antiemulation, etc. So, it is very expensive for program analysis to extract the cryptographic algorithm from IoT apps because of these protections.

Based on the fact that nowadays most apps use a standard cryptographic algorithm for encryption and signature, Zuo [37] utilizes API hooking to extract the cryptographic algorithm. He hooks 61 well-defined cryptographic APIs in Android SDK to intercept their arguments and return values, then reconstructs the control flow and data flow based on the API hooking log, and reexecutes them out of the app. After analyzing a lot of IoT apps, we find that this method lacks flexibility, because it just supports well-defined cryptographic APIs. In our running example, the signature is generated as *sign* ≔ *MessageDigest.getInstance("MD5").dgest(Base64.encodeToString("...password=e10adc3949ba59abbe56e057f20f883ecode=7496phone=1383815∗∗∗∗..."))*. Obviously, *getInstance* and *dgest* will be logged as they are well-known and being hooked, but the arguments will not be logged as *encodeToString* is not well-known and not being hooked. So, the data flow will be interrupted and the cryptographic algorithm for generating *sign* cannot be reconstructed from the API hooking log. In addition, private cryptographic algorithm is also not supported by this method.

In fact, our final goal is the output, not the code of the cryptographic algorithm. If we treat the whole IoT app as a cryptographic algorithm, user input as the arguments, the output as the password reset message, then we can utilize UI automation [38] to input every possible SMS code and request password reset, app will execute its code to calculate the *sign*. So, we can generate a cryptographically consistent message without extracting the cryptographic algorithm. Moreover, this approach is independent on the cryptographic algorithm, so it does not have the drawbacks of program analysis and API hooking, it can support complex app logic, code obfuscation, private cryptographic algorithm and so on. Based on UI automation, we can generate cryptographically consistent messages when mutating the SMS code, but we still need to address several other implementation challenges for performing brute-force attack test automatically.

3.3. Implementation Challenges and Solutions. There are three implementation challenges for utilizing UI automation to perform brute-force attack on SMS code automatically: (1)

identifying password reset UI, (2) identifying interesting parameter, (3) time expiration on the SMS authentication code, and (4) unnecessary brute-force requests. We must address all of them.

Identifying Password Reset UI. Before we can use UI automation to input phone, SMS code, new password to drive the app to generate the password reset message, we need to identify the password reset UI firstly. Unfortunately, there is no straightforward information to declare where is the password reset UI. We analyzed a lot of apps and found that most of password reset UI contain the common feature: (1) an editable UI component with a default description like *input your phone number* for guiding users to input phone, (2) an editable UI component with a default description like *input SMS code* for guiding users to input SMS code, (3) an editable UI component with a default description like *input new password* for guiding users to input password, and (4) a clickable button with a default description like *confirm* for guiding users to submit request. This is because developers usually need human-friendly text information to guide users to input password reset parameters in the right component. So, we can enumerate every UI of an IoT app and check which one contains this common feature to identify the password reset UI. In addition, some apps divide the logic of password reset into several parts, so they will use more than one UI to guide users to input all parameters. We can analyze a set of sequential UI to identify the root of password reset UI. Details about identifying password reset UI are presented in Section 4.1.

Identifying Interesting Parameters. Usually, there are more than four parameters in password reset message. But interesting instances are *phone, code, password, sign*, we need to identify them. Messages based on REST usually encodes a parameter as a key-value in the URL or as JSON/XML in the content [39, 40]. If the parameter key name is well-known, such as *verifyCode* and *password* used in our running example, it is easy to identify them by regular expression matching. But different apps can use different key names, *checkCode, vCode, ck* are also acceptable for the SMS code, so it is challenging to identify them automatically. However, the input of IoT apps is controlled by us via UI automation, so we can use two different values for a parameter and analyze the two corresponding messages to identify the parameter (e.g., in our running example, we input *code1* and *code2* for the *code*, we can find both values in the messages and infer the key name for SMS code is *code*). Details about identifying interesting parameters are presented in Section 4.2.

Time Expiration. IoT clouds always assign an expiration limit (varies from 2 ~ 30 minutes) on the SMS code. The brute-force test must be as fast as it can, or the authentication code will become unusable before being mutated to the right one. If the password reset message is not encrypted or contains no signature, we can mutate code based upon a captured message directly and send mutated messages to the cloud at a fast speed. Or we move to use UI automation to generate cryptographically consistent messages. In fact, the speed of UI automation is very slow, if we serially mutate the code and request the cloud in a real-time environment, we will meet the expiration. However, we can use an offline environment to record all request messages and then replay all of them to the cloud via a high-performance computer. Because these request messages contain all possible values for the SMS code, so the password will be reset successfully. Details about time expiration are presented in Section 4.3.

Unnecessary Requests. During the attempts of password reset with mutated codes, if the account password is reset successfully, we can discard the rest of messages to reduce unnecessary requests. But the response contents are diverse, there are no standards and documents for whether the password reset response is successful or not. We observed that the response usually contains a status message such as *your code is invalid* or *too many unsuccessful attempts*. Different messages have a different length, this will make the length of the whole response message different. So, we can monitor the response length. If it is changed, it means a new status, such as *success of password reset* or *being blocked by the cloud*. Then we try to login the account with the predefined password, if success and we discard the rest messages, if failure and we infer this IoT device account is not vulnerable.

By addressing above challenges, we can use UI automation to generate cryptographically consistent message while mutating the code to all possible values. In addition, if there is no signature or encryption in password reset message, we can directly mutate the code based on a captured password reset message to perform brute-force attack test.

3.4. SACIntruder. To check whether a given IoT device is vulnerable to the SMS-base authentication, we design a tool named *SACIntruder*, to automatically perform brute-force attack on the SMS code. The only knowledge we need is the phone number that is used to register as the device account. PMitM [21] presented a method about how to get victim's phone, and how to get the phone is beyond the scope of this paper. Moreover, IoT app may choose a binary protocol to build a request message. We only focus on HTTP protocol here, because we observed that a large number of Android apps use REST API that is built on the HTTP. We consider HTTPS as HTTP, because HTTPS uses SSL as its lower transport protocol and we can use a self-signed certificate to bypass HTTPS.

The whole progress of our brute-force attack supported by *SACIntruder* is presented in Figure 1. There are four steps: (1) *SACIntruder* starts every activity of an IoT app to identify password reset UI. (2) *SACIntruder* drives the app to input victim phone and request for SMS code, then it replaces the victim phone with a test phone via message interception, this replacement will prevent the cloud send a code to the victim at the very beginning. (3) *SACIntruder* drives the app to input parameters for password reset and request the cloud, then it identifies interesting parameter. If the message contains encryption or signature, it will stop forwarding the following message to the cloud, and continue driving the app to input all possible SMS code and save all messages into a database.

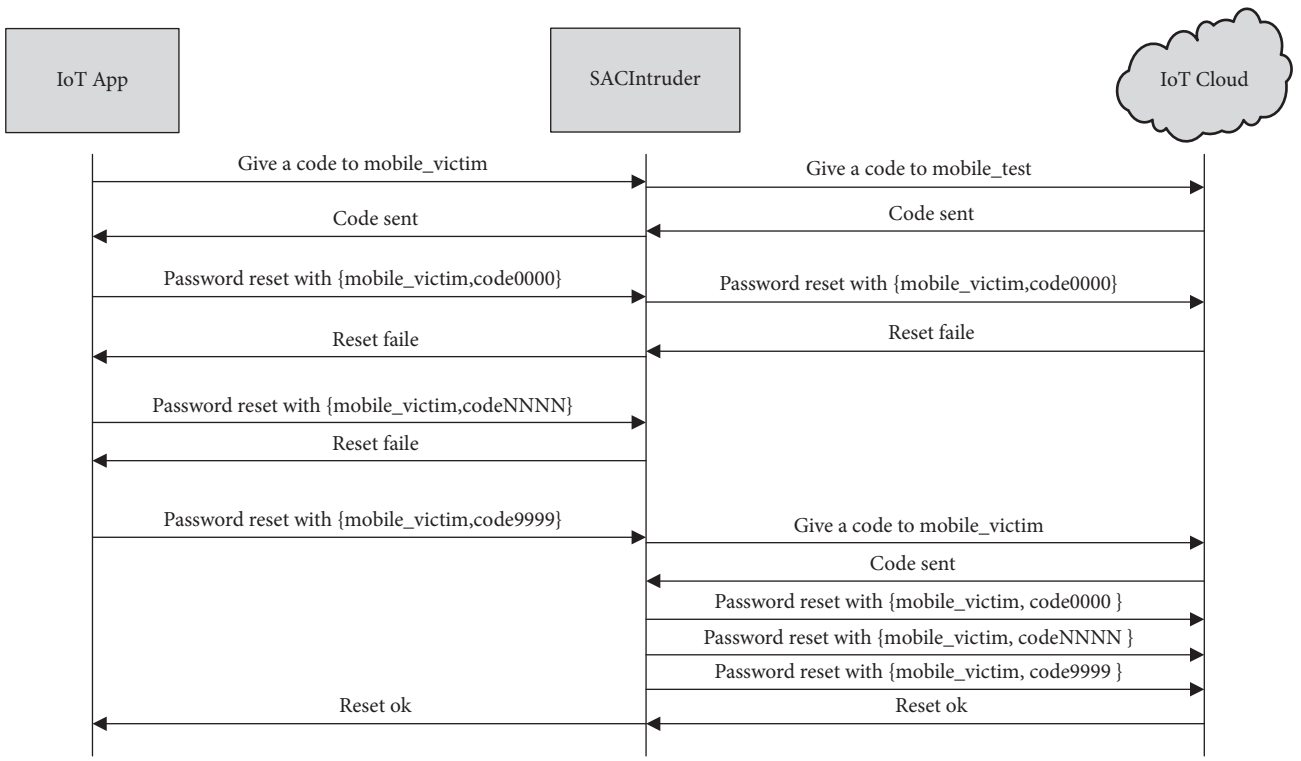

Figure 1: Password reset via brute-force attack to SMS authentication code: code0000 is a SMS authentication code whose value is 0000 and codeNNNN is an instance of all possible SMS authentication code whose value ranges from 0000 to 9999.

(4) At last, *SACIntruder* begins to replay the message for SMS code, and all messages for password reset to the cloud. If the account is vulnerable, its password will be reset successfully.

An overview of *SACIntruder* is presented in Figure 2: its inputs are an IoT app and a victim phone and the output is whether the IoT app is vulnerable. There are three key components:

(i) **MessageGenerator** using UI automation to control the execution of IoT app, it performs static analysis and dynamic analysis to identify UI and drives app to generate cryptographically consistent messages.

(ii) **ParameterIdentifier** using different inputs to identify parameters in the password reset message, it contains a network proxy to intercept the communication between the app and cloud. It cannot input data to the app, so it communicates with MessageGenerator to achieve this goal.

(iii) **RecordReplayer** uses a database to record all password reset messages and replay them to the cloud parallelly at last; it is running on a high-performance computer for fast speed.

4. Detailed Design

In this section, we present the detailed design of the three key components of SACIntruder. We first describe how does *MessageGenerator* generate a password reset message in Section 4.1, then explain how does *ParameterIdentifier* identify interesting parameters in Section 4.2, and then present how does *RecordReplyer* bypass the time expiration on SMS code in Section 4.3.

4.1. Password Reset Request Message Generation. MessageGenerator is responsible for password message generation, it utilizes UI automation to control the execution of IoT apps. First, it runs the app and identifies the password reset UI by enumerating every activity and checking the common feature in Section 3.3. As stated previously, there are two password reset UI models. Figure 3 is the classic UI model, named *Single-stage Password Reset*, because all parameters are inputted in a single UI. And Figure 4 is another model, named *Multistage Password Reset*, because all parameters are divided into two or three groups and inputted in several UI. Then, *MessageGenerator* enters the right UI, inputs parameters, and triggers request to generate password reset messages.

4.1.1. Single-Stage Password Reset Request Message Generation. For the *Single-stage Password Reset*, all UI elements used to receive these parameters are defined in an activity. First, we identify the UI via analyzing every activity independently and then input every parameter to this activity to drive the app to generate password reset messages.

Static Method to Identify UI. In Android, every activity must be declared in a file named *AndroidManifext.xml* that can be

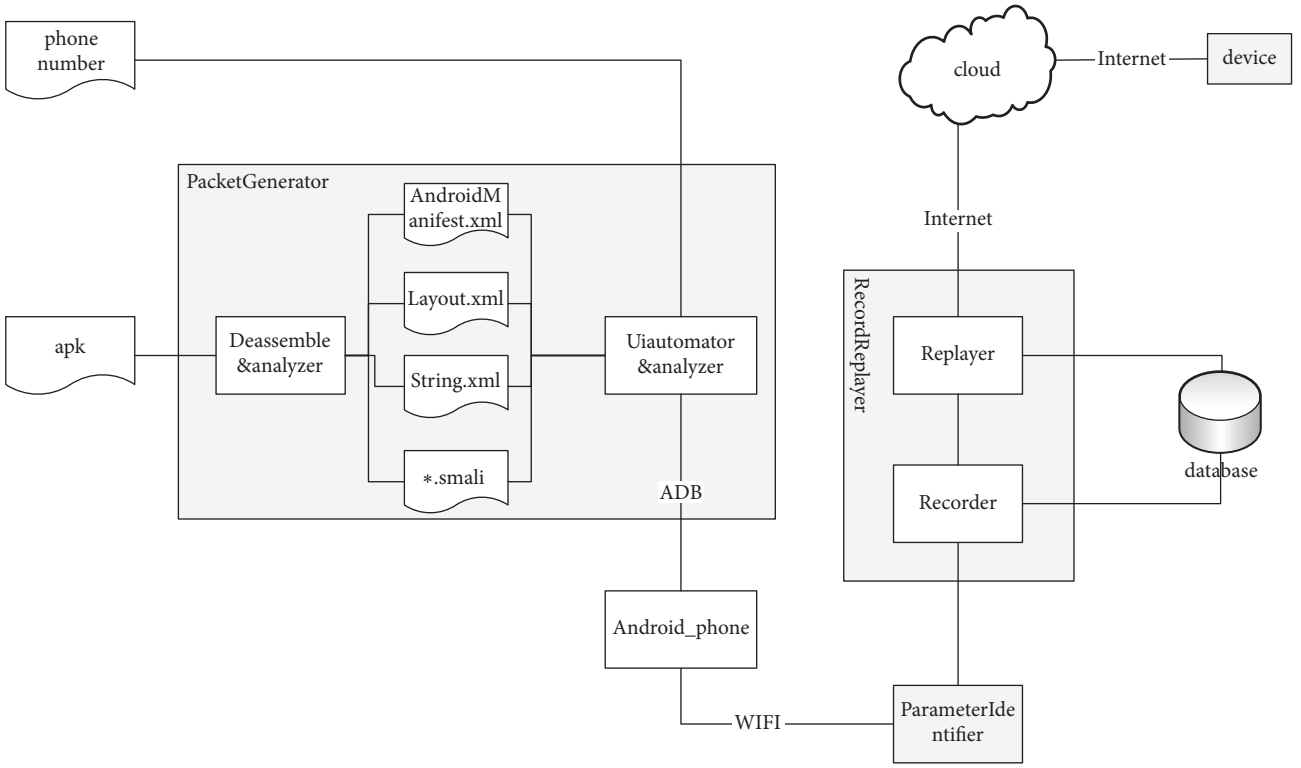

Figure 2: An overview of SACIntruder.

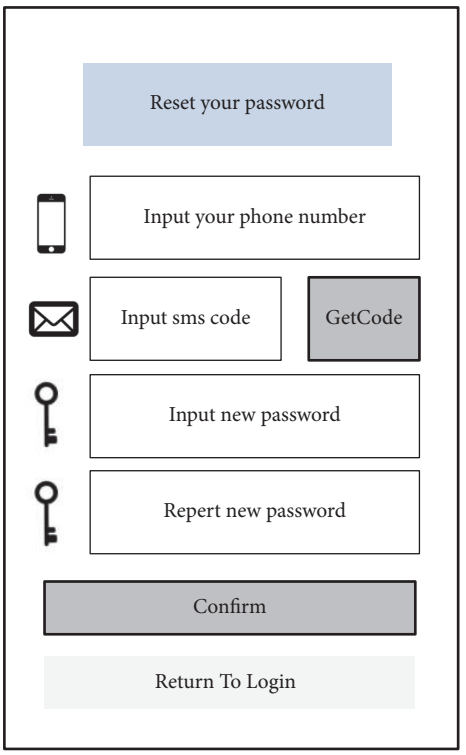

Figure 3: Single-stage password reset UI model.

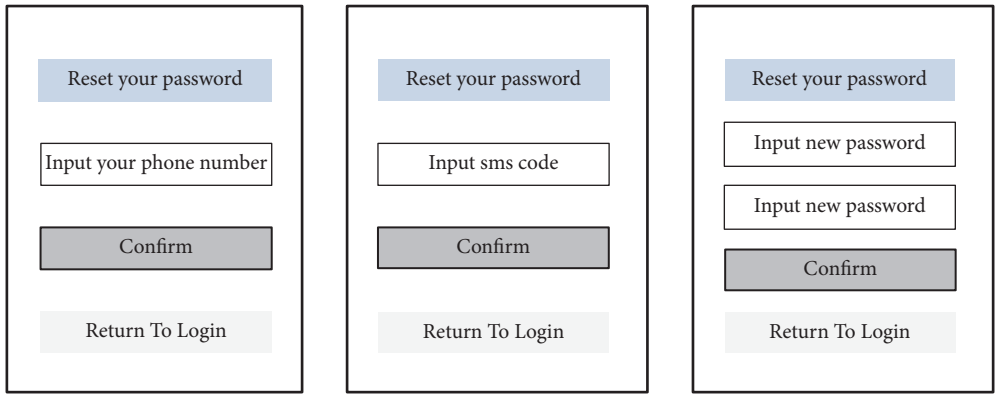

Figure 4: Multistage password reset UI model.

extracted from the app binary file. Some other works such as AppsPlayground, SMV-Hunter, and Gui Ripping [41–43] use this manifest to start dynamic UI exploration. But dynamic exploration is usually slow, so we first use static method. For an activity, all UI elements (e.g., *Buton, ImageButon, CheckBox*) are defined in a layout, and the activity uses *setContentView* to load the layout. In theory, a layout can be a standalone file or a piece of codes about dynamic layout generation. After analyzing a lot of apps, we found that most apps use a standalone file. So, we analyze the layout file of an activity to infer whether it is the right UI via checking the common human-friendly information.

Because an activity loads its layout by calling API *setContentView*, we need to analyze the code of an activity to get its layout. The source code of an IoT app is always not available, but it is not a problem. Because Android app is very similar to the Java archive, it is very easy to disassemble the app. We can use *apktool* [44] to disassemble the app. Every part of the app can be extracted to basic elements, such as string pool files, code files for every class, layout files, image files and so on. In these elements, a file named *public.xml* is very important, because the disassembled code of an activity will not use a name to reference its layout but an integer number that is defined in this file. Another file named *strings.xml* is also important, because the human-friendly information in password reset UI are usually defined here. There are in total five major steps in order to identify the right UI via static method:

(i) Disassemble an IoT app with *apktool*. If the app is not protected by a packer [45], we can get every part of the app. Otherwise, we can only get a subset of all parts, and we will move to dynamic method.

(ii) Get the name list of all activities from the manifest. Again, every activity must be declared in this file. If the app contains a password reset activity, we can analyze every activity to find it out. In addition, we get the package name of the app from the manifest, combine package name and activity name to generate the full name of the activity code file. The output of this step is an array of *activityFullname*.

(iii) Find the layout via disassembled code of every activity. The entry point of an activity is a callback function named *onCreate*. We scan the disassembled code of every activity's *onCreate* procedure to get the referenced layout. Again, the layout in the disassembled code has been converted to an integer. The output of this step is an array of *(activityFullname, layoutInteger)*.

(iv) Get the layout filename via *public.xml*; Android uses this file to map the integer to a name. The output of this step is an array of *(activityFulname, layoutName)*.

(v) Analyze the referenced layout file to infer whether it is the password reset activity. A layout file is an XML file that contains every UI element definition with type name and default value. Figure 3 is a typical *Single-stage Password Reset* UI that contains three input elements, two submit elements, and some human-friendly strings. If the layout file contains elements definition like this, we can infer it is the right UI. The output this step is *activityFullname*.

Dynamic Method to Identify UI. If an IoT app is protected by a packer, the real code and layout will be hidden from static analysis except for the manifest. But manifest just contains names of all activities, we cannot get the layout referenced by an activity from it. So, we move to dynamic method.

In Android, every running activity is managed by the activity manager. All activities are maintained via a stack, and they are arranged in the order according to the time when each activity is opened and only the top activity in the stack is painted on the screen [46]. To paint the screen, activity manager keeps a screen layout that contains the top activity and other UI elements such as system virtual home key. Moreover, Android allows dumping the layout of the current screen to support debugging. So, we can run every activity of an app and dump the current screen layout to get the layout of the running activity without regarding the packer or dynamic layout loading. There are in total four major steps in order to identify the right UI via dynamic method:

(i) Get all activity names from the manifest. This step is the same as the previous static method.

(ii) Start every activity of the IoT app. This will bring an activity to the top in the system activity stack. We use an Android phone and turn its debugging support on, then we use command *am start package/activity* to start an activity.

(iii) Dump the current screen layout. There is an android shell command named *uiautomator* [47] that is designed for dumping layout of the current screen. We use it to dump the layout and save it as a file in the phone, then we use Android Debug Bridge [48] (ADB for short) command to download the layout file from the phone.

(iv) Analyze the layout to infer whether it is the password reset UI. Unlike traditional standalone layout, screen layout already loaded the values of associated strings that are defined in another file. Another difference is it may contain some UI elements which do not belong to the app such as the status bar on the top of current screen or a popup window. To reduce the noise, we first exclude these elements, and then we check the common feature used in the static method. If found, we infer this activity is the password reset UI.

Contrast to the previous static method, dynamic method is much slower, because it runs the app and uses debug interface to get the key information. But it is very useful when the app is protected by a packer [45], or the app is implemented to load the activity layout dynamically via code.

Generate Password Reset Message. After identifying the password reset UI, we can use UI automation to input parameters for driving the app to generate a password reset message. The method is straightforward: enumerating parameter elements in the layout, inputting data and clicking the submit button to generate the message. Some apps contain one password elements, while two password elements for some other apps. It is easy to identify this difference, because the two elements for password are very similar based on their description text.

To capture the messages, we can use Wireshark, tcpdump and libpcap [49, 50]. However, we just focus on the application layer content that is consumed by the app and the cloud. We found that Android supports WIFI proxy that means all HTTP/HTTPS requests from app can be redirected to the WIFI proxy. Meanwhile, there are plenty of open-source HTTP proxy servers, we can utilize them to analyze the content of the HTTP layer. So, we use the WIFI proxy of Android and HTTP proxy server to capture the generated message.

4.1.2. Multistage Password Reset Request Message Generation. Many IoT apps use *Single-stage Password Reset*, but some apps use *Multistage Password Reset*. We directly use dynamic analysis to identify this password reset UI, because we cannot infer it by analyzing a single activity. Then we input parameters to drive the app to generate password reset message.

Identify the UI. A sample of *Multistage Password Reset* model is presented in Figure 4. There are three activities, the first one just receives the phone number, if it is confirmed, the app will switch to next one and an SMS code request message will be sent to the cloud; the second one receives the SMS authentication code, if confirmed, the app will switch to next one; the third one receives the password, if confirmed, the password reset message will be sent to the cloud.

It is challenging to recognize this type of password reset UI, because every single activity does not contain sufficient information. But it is obvious that if there are some sequential activities in the app, and every one of these activities contains a different part of the whole parameters for password reset, we can infer it is the right UI. So, we use a state machine:

(i) First, we define a 3-bit-vector, the first bit means the presence of a phone number UI element, the second bit means the presence of an SMS code, the third bit means the presence of a password. And we define four states: *s0* is the initial state, *s1* means that phone number has been met, *s2* means that phone and SMS code have been met, and *s3* is the final state that means all parameters have been met.

(ii) Second, we run every activity and dump the current screen layout. Then we check the presence of phone number, SMS code, and password, and change the state according to the current state.

(iii) Third, if we reach the *s3*, we infer the activity that changes the state to *s1* is the password reset UI.

Generate Password Reset Message. Here we use the same method used in the *Single-stage Password Reset* model to generate the messages and capture them. The only difference is we need to input parameters in several UI, because each UI just receives a part.

4.2. Identifying Interesting Parameter in Password Reset Message. Usually, there are more than four parameters in the password reset message. But we are just interested in *phone, code, password, sign*. We need a principled approach to parse the request message and identify our interesting parameters.

4.2.1. Parsing Parameters in Request Message. Again, we focus on IoT app built on REST [51] that uses HTTP as its transport protocol. According to the HTTP protocol specification, each request message consists of (1) a request line (e.g., *GET /ci/user/getVerify Code?uid=-1&phone=1383815∗∗∗∗&imei= HTTP/1.1*), (2) extra request-headers (e.g., *Host: ∗ ∗ ∗. ∗ ∗∗house.com.cn*), (3) an empty line, and (4) optional message body. There are two main methods for HTTP request, *GET* and *POST*. For a *GET* request, it can host a small piece of data in the URL. For a *POST* request, it can host a lot of data in the message body. And both methods are popular in IoT apps.

According to the URL specification [52], data can be encoded as a *k-v* block in the URL parameters. The first *k-v* is connected to the base URL via symbol "?", *k-v* is connected via the symbol "=", and each *k-v* pair is concatenated via "&". To parse these parameters, we first parse the path segment via scanning the reserved path symbol "?" to get the first *k-v* (*uid=-1* in our running example). Then we use "&" to parse more *k-v* (*phone=1383815∗∗∗∗* and *imei=*). It is quite

straightforward to index the parameter name and its value, and we store them in a pair *(name, value)*.

Parameters can also be in the message body when the method is *POST*. There is no limitation on the format of the body, because requester can specify its format type via a *Content-Type* request header. But the best practice of REST development suggests developers to use JSON and XML as the format for message body and we only parse these two popular formats. JSON and XML have a hierarchy tree structure, which means that each value can be tracked by the path from the root of the tree. And we still store them in a pair *(name, value)*.

4.2.2. Identifying Interesting Parameter with Different Input Value. After parsing parameters, we need to know whether a parameter is interesting, such as the signature. We do not care those uninteresting parameters (e.g., *app version* and *mobile version*), because they are irrelevant to the procedure logic of password reset and we just keep their original values. There are four interesting parameters: *phone, code, password, sign*. For the phone number, it is used in SMS code request message and password reset request message. SMS code is the most important parameter; we want to mutate it to all possible values. Signature is a hidden parameter that is automatically generated by the app logic. If signature or encryption is found, we utilize UI automation to drive the app to generate cryptographically consistent messages.

The values of these parameters are from the user input in the password reset UI, and we can control the input value via UI automation. So, we change the input value of a UI element with different values, then we analyze the generated messages to locate the two different input values. By this way, we infer the interesting parameters without the knowledge of their key name:

(i) For the *phone*, we input two different phone numbers, same code, and same password in the password reset UI. Then we trigger the UI to send the password reset message and parse these two messages. If a parameter with the same key contains both different input values, we infer the parameter is the phone number. For our running example in Boxes 1–4, we use two inputs *(phone1, password, code)* and *(phone2, password, code)* to drive the app to generate two messages. In the following analysis, we can find a parameter whose key name is *phone* contains our two different values *(phone1, phone2)*. So we get the phone parameter without any knowledge about its key name.

(ii) For the *code*, we input two different SMS codes, same phone number, and same password. Then we infer the SMS code like the phone number.

(iii) For the *sign*, we input two different SMS codes, same phone number, and same password. Unlike the SMS code, we do not have any input value for signature, so we do not know what should be searched in the two messages. Cryptographic algorithms have a desirable property named avalanche effect [53], it means that a little difference in the inputs will cause a dramatic difference in the outputs. Signature is built on a cryptographic algorithm, and we can use this common property to infer the *sign*. We parse these two messages, if we find a parameter that has a very high score of difference in its two values, we infer it is the *sign*. We use Euclidean distance [54] as the score and we also use this method to check whether the whole message is encrypted.

4.3. Time Expiration Bypass. We utilize UI automation to generate cryptographically consistent messages and support all types of algorithm including private versions. But it is very slow for UI automation, because every password reset request will indeed (1) get parameters from the UI elements and check validation, (2) perform encryption or signature, (3) create a new TCP connection to the cloud, (4) build an HTTP request message containing the data from step2, (5) send the message to the cloud, (6) wait for the response message and parse it, and (7) synch of UI events. In addition, for an Android phone, capability for computing and networking is very limited. So, if we directly use UI automation to drive the app to perform brute-force on the SMS authentication code, we will fail because of time expiration of the SMS code.

After analyzing a lot of IoT apps, we found that we can use an offline-style method, the core idea is that we just drive the app to generate all messages in an offline environment and replay all of them in an online environment:

(i) Intercept the SMS code request message from the IoT app and record it, replace the phone number by another one used for experimentation, forward the modified message to the cloud, receive the response message and record it, at last forward the response to the app. Then the app can send password reset message.

(ii) Intercept the password reset request message from the IoT app, forward it to the cloud if it is the first one, receive the response and record it, forward the response to the app. In the following requests, we will not forward the requests to the cloud and use the recorded response message to emulate the cloud response. In this way, we generate all password reset messages in an offline environment without communication to the cloud.

(iii) After all possible messages have been generated, we perform the brute-force test. First, we replay the recorded SMS code request message to the cloud to generate an SMS code for the real phone number. Second, we replay all password reset messages to the cloud. We use socket pool and thread pool to maximize the speed. And we monitor the response length, if it is changed, we stop replaying and use the predefined password to try account login. If successful, we infer the IoT device account is vulnerable.

5. Evaluation

We have implemented *SACIntruder* based on several open-source tools, our message generation is built on *uiautomator*

Table 1: Summary of IoT Apps under testing.

Type	Vendor	AndroidApp	UI Model
Watch	ToyCloud	com.watch∗ ∗ ∗.www	Single-stage Password Reset
Lock	Panasonic	com.∗ ∗ ∗.digitallock	Single-stage Password Reset
SharingCar	Panda	com.∗ ∗ ∗.usecar	Single-stage SMS Login
SharingCar	win-sky	com.∗ ∗ ∗.drivevi	Single-stage SMS Login
Router	ximo	com.∗ ∗ ∗.router	Single-stage Password Reset
HomeGate	HuiJu	com.∗ ∗ ∗.devices	Single-stage Password Reset
Robot	lejurobot	com.∗.zelos	Multi-stage password Reset
Car Cmera	DUDU	com.∗ ∗ ∗.∗ ∗ ∗.laucher	Single-stage Password Reset
HomeGate	BroadLink	com.∗ ∗ ∗.rmt	Multi-stage password Reset
Car	DasAuto	com.∗ ∗ ∗.faw.vw.∗ ∗ ∗	Multi-stage password Reset
IP Camera	uniview	com.∗ ∗ ∗.ezview	Multi-stage password Reset
Car	DongFeng	com.∗ ∗ ∗.windlink	Single-stage Password Reset

Table 2: Summary of discovered vulnerabilities.

AndroidApp	CNVD	Public
com.watch∗ ∗ ∗.www	CNVD-2017-02059	Yes
com.∗ ∗ ∗.digitallock	CNVD-2017-03908	Yes
com.∗ ∗ ∗.usecar	CNVD-2017-04583	Yes
com.∗ ∗ ∗.drivevi	CNVD-2017-06343	Yes
com.∗ ∗ ∗.router	CNVD-2017-15081	Yes
com.∗ ∗ ∗.devices	CNVD-2017-03909	Yes
com.∗.zelos	CNVD-2017-01003	Yes
com.∗ ∗ ∗.∗ ∗ ∗.laucher	CNVD-2017-09696	Yes
com.∗ ∗ ∗.rmt	CNVD-2017-12023	Yes
com.∗ ∗ ∗.faw.vw.∗ ∗ ∗	CNVD-2017-25143	Yes
com.∗ ∗ ∗.ezview	CNVD-2017-12075	Yes
com.∗ ∗ ∗.windlink	CNVD-2017-15147	Yes

[55] and *mitmproxy* [56], our RecordReplayer deploys *SQLite* [57] as its persistent data storage. And we wrote python code to drive UI automation and intercept messages, we wrote C code to replay recorded messages.

5.1. Experiment Setup

IoT Devices. We selected 12 representative IoT devices from different categories, including car, sharing car, robot, smart lock, smartwatch and smart router, etc. All these devices have an official Android app used to manage them. The detailed specifications of these IoT apps are described in Table 1. In particular, we summarize app information and their UI model. There are three types of UI model, *Single-stage Password Reset*, *Single-stage SMS Login*, *Multistage Password Reset*. *Single-stage SMS Login* can be considered as a special type of *Single-stage Password Reset*.

Testing Environment. Our IoT UI automation runs on an Ubuntu 12.04 PC with Intel Core i7 quad-core 3.6 GHz CPU with 16G RAM, a wireless router *TP-LINK TL-WAR1200L 1200M* and an Android phone *OnePlus*. Both the phone and PC are connected to the same wireless router, the WIFI proxy of the OnePlus is configured to the PC. We did not test IoT account of other customers, we register an account with our experimental phone to simulate the victim account. During our testing, the SMS authentication code will be sent to our phone, but we never use it and we just perform brute-force on the SMS authentication code.

5.2. Evaluation Result. We found the official apps of these devices are vulnerable to the brute-force attack on SMS code, an attacker can steal the accounts of these devices to control them remotely. As shown in Table 2, we found 12 zero-day vulnerabilities, the third column indicates whether the vulnerability can be indexed publicly in the *China National Vulnerability Database* [58](CNVD for short). Again, all vulnerabilities we founded have been reported to CNCERT/CC [7] to help the vendor fix them.

There are 8 vulnerabilities about *Single-stage Password Reset and Login*, and six of them are about password reset. There are other 4 vulnerabilities about *multistage password Reset*. Our tool supports both single-stage UI model and multistage UI model.

5.3. Case Studies. **CNVD-2017-03908** is a password reset vulnerability about a smart lock that belongs to Panasonic.

First, our tool finds that the app uses *Single-stage Password Reset* model. Second, it drives the app to generate a password message. Third, it finds that there are no encryption and signature in the message, so it mutates the SMS code in the message directly on a computer that is faster than a smartphone. Fourth, the tool sends mutated messages to the cloud to reset password successfully. An attacker can use this vulnerability to steal the victim account, then he can open victim's door to do anything. Panasonic has fixed this issue now.

CNVD-2017-04583 is a login vulnerability about a sharing car that belongs to a company that owns many cars and leases them to the customers. First, our tool drives the app to generate a password message. Second, it finds that the message has a signature, it cannot directly mutate the SMS code. Third, it uses UI automation to drive the app to try all possible code to generate password messages and record them to a database. At last, it replays all messages to the cloud to generate a login token and forwards the token to the application. As a result, an attacker can use the car sharing service in victim's name. It has been fixed now.

CNVD-2017-15147 is a password reset vulnerability about a car intelligent interconnected system that can send commands to the *Electronic Control Unit* (ECU for short) to open/close door, window and *Car-Carrying Air-Conditioning*. The app also uses *Single-stage Password Reset* model, and the SMS code parameter uses a strange key name *checknum*, not traditional $***code$. Our tool can find it because we use different input values to infer parameter. An attacker can utilize this vulnerability to drive a car away. It has been fixed now.

CNVD-2017-12023 is a password reset vulnerability about a smart home controller that can control a lot of devices provided by broadlink, the app uses *multistage password reset* model. Our tool first finds the right UI by analyzing several sequential activities and drives the app to generate a password reset messages. Then our tool finds that the message is encrypted via inputting different values, so it uses UI automation to generate all password messages and record them to a database. At last, it replays all messages to the cloud to reset password successfully. An attacker can use this vulnerability to control home devices. It has been fixed now.

6. Discussions

Possible Countermeasures to Prevent This Attack. This paper shows that *Reset Your Password* of IoT apps may be vulnerable, because an attacker can brute-force attack SMS authentication code to crack IoT device user account without any victim's interaction. The core insights are (1) the search space of SMS code used in the password reset is much smaller than password, (2) the cloud does not limit the number of attempt for account management. Frequency limitations, such as IP-based strategy and Account-based strategy, may pose some problems. If an attacker and legal customers are behind the same NAT [59] gateway, IP-based strategy will block legal customers to access to their devices. Account-based strategy will block the device owner to access the device, if an attacker performs brute-force on his account. The best protection is to deploy *CAPTCHA* [59] in password reset message, because our method relies on pregeneration of all password reset messages containing every possible SMS code. To balance security and usability, we can just activate *CAPTCHA* when the number of unsuccessful attempts meets a threshold.

Can SACIntruder Works on iOS. SACIntruder can also be implemented on iOS, because the core insights are (1) *Reset Your Password* is a feature at application level, it is independent on the low-layer smartphone operating system. (2) Human-friendly information, such as "input your phone" in Reset Your Password UI, is designed to guide user to input easily. We can use this to identify the UI, no matter the app is running on Android or iOS. (3) SMS authentication code is also independent on the low-layer smartphone operating system. It is usually a 4-digital or 6-digital number. Its search space is small. The major difference in these two platforms is UI automation, because it is dependent on the low-layer smartphone operating system. On Android, we can use uiautomator to control third apps. But we cannot do this on iOS because of its app sandbox, we need a jailbroken iphone to bypass this limitation. Then we can write our code to control the whole iOS, such as capturing the screen to identify UI via image processing and inputting data to third apps.

Limitations. Our paper just focuses on HTTP/HTTPS protocols because of the popularity of REST. But there is no limitation for IoT apps, they can use any protocol even private version based on binary format. Our tool SACIntruder uses the WIFI proxy of Android phone to help packet interception, so an IoT app can detect the WIFI proxy to prevent the packet interception.

7. Related Work

Vulnerability Discovery in Embedded/IoT Device. Costin [60] used static analysis to analyze more than 30000 firmware images to find bugs including XSS, hardcoded private key-pairs and back-door. Cui and Stolfo [61] found more than 500000 publicly accessible devices containing default credentials via Internet scanning. Cui and Costello [62] found that remote firmware update functionality can be exploited by attackers to insert malware. Davidson [11] used KLEE symbolic execution engine to detect memory vulnerabilities in open-source firmware. Li [63] ported the QEMU emulator to detect vulnerabilities in SoC. Zaddach [42] combined emulator and a real device to detect vulnerabilities. Chen [13] ported QEMU to run the Linux-based firmware to detect vulnerabilities on a large scale. Wang [64] designed a fuzz framework RPFuzzer by sending normal network packets and monitor CPU and system logs to detect vulnerabilities in routers. Costin [14] analyzed the management web interface in devices to detect vulnerabilities. Chen [65] designed a fuzzing framework named IoTFuzzer that uses the rich protocol information in IoT official app to guard fuzzing. In contrast to these works, our paper focuses on vulnerability on the IoT account. If the account is vulnerable, an attacker can use the account to control device via the account.

Account Security. SQL injection, XSS, CSRF, and logic fault are usually being used to hack an account. Halfond [66] presented an extensive review of the different types of SQL injection attacks. Vogt [67] tracked the flow of sensitive information inside the web browser to prevent XSS. Barth [68] performed an experimentation about CSRF vulnerability in 283945 advertisements and presented a new variation on CSRF attacks. Dalton [69] presented a novel methodology based on Dynamic Information Flow Tracking to mitigate authentication and access control vulnerabilities. Pellegrino [70] used a black-box methodology to detect logic flaws in web applications based on automatic identification of a number of behavioral patterns. Wang [71] performed a security analysis of cashier-as-a-service based web stores and found several logic flaws that can allow an attacker to buy an item at an arbitrarily low price. Gelernter [21] presented the password reset MitM attack that exploits the similarity of the registration and password reset processes, this attack can be used to take over user accounts. Zuo [37] performed password brute-forcing via automatic forgery of cryptographically consistent request message, it hooked standard cryptographical API to get knowledge of algorithms. Contrast to these works, our paper focuses on the SMS authentication code for account management, an attacker can perform brute-force attacking on the code to steal IoT account without any interaction of victim.

Mobile App Analysis. Monkey [72] is a testing tool for dynamic exploring the app UI automatically. Machiry [73] proposed a system named Dynodroid for generating relevant inputs to unmodified Android apps. Rastogi [41] proposed a framework that automates the analysis of Android application, it integrated multiple components comprising different detection and automatic exploration techniques. Anand, Mirzaei and Zuo applied symbolic execution [74–76] to perform more systematic dynamic analysis, so they can retrieve more internal knowledge but heavy overhead. Cui [77] proposed a tool named Discoverer for automatic protocol reverse engineering from network traces. Beddoe [78] maintained a protocol informatics project, it is very useful for protocol reverse engineering. Our paper is particularly by Monkey and protocol informatics project, we use UI testing tool to explore the UI component and input test data, we use different input values to identify interesting parameters.

8. Conclusion

We have performed the first security study of *Reset Your Password* that is popular in IoT device account and found it may be vulnerable because of the SMS-based authentication. We have presented the design, implementation, and evaluation of SACIntruder, a tool that is able to automatically perform brute-force attack on SMS code to test whether an IoT account is vulnerable. We have tested SACIntruder with representative IoT devices from different categories, including car, sharing car, robot, smart lock, smart watch, and smart router and found 12 zero-day vulnerabilities. We reported all vulnerabilities to CNCERT/CC to help the vendor to fix them, and all of them have been fixed now.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request. The vulnerabilities found in this paper can be accessed in the CNVD.

Disclosure

A conference version of this paper was presented at the RESEC 2018.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported in part by the Science and Technology Project of State Grid Corporation of China, National Natural Science Foundation of China (Grant no. 61572115), and National Key Research and Development Plan (2017*YFB*0802900), and Project 2117*H*14243*A* and Sichuan Province Research and Technology Supporting Plan, China.

References

[1] Google. Android wear, 2018.

[2] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2016*, pp. 461–472, Xi'an, China, June 2016.

[3] C.-L. Hsu and J. C.-C. Lin, "An empirical examination of consumer adoption of Internet of Things services: Network externalities and concern for information privacy perspectives," *Computers in Human Behavior*, vol. 62, pp. 516–527, 2016.

[4] J.-Y. Lee, W.-C. Lin, and Y.-H. Huang, "A lightweight authentication protocol for Internet of Things," in *Proceedings of the 3rd International Symposium on Next-Generation Electronics, ISNE 2014*, twn, May 2014.

[5] C. Doukas, I. Maglogiannis, V. Koufi, F. Malamateniou, and G. Vassilacopoulos, "Enabling data protection through PKI encryption in IoT m-Health devices," in *Proceedings of the 12th IEEE International Conference on BioInformatics and BioEngineering, BIBE 2012*, pp. 25–29, November 2012.

[6] X. Wang, J. Zhang, E. M. Schooler, and M. Ion, "Performance evaluation of Attribute-Based Encryption: toward data privacy in the IoT," in *Proceedings of the proceedings of the 2014 1st IEEE International Conference on Communications (ICC '14)*, pp. 725–730, Sydney, Australia, June 2014.

[7] CNCERT/CC, National computer network emergency response technical team/coordination center of china, 2018.

[8] Lucian Constantin, Hackers found 47 new vulnerabilities in 23 iot devices at def con. CSO, 2016.

[9] Chris Brook, *Travel Routers, Nas Devices among Easily Hacked iot Devices*, 2017.

[10] N. Zhang, S. Demetriou, M. Xianghang et al., "Understanding iot security through the data crystal ball: Where we are now and where we are going to be," https://arxiv.org/abs/1703.09809.

[11] D. Davidson, M. Benjamin, T. Ristenpart, and J. Somesh, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the In USENIX Security Symposium*, pp. 463–478, 2013.

[12] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "FirmUSB," in *Proceedings of the the 2017 ACM SIGSAC Conference*, pp. 2245–2262, Dallas, Texas, USA, October 2017.

[13] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA.

[14] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2016*, pp. 437–448, chn, June 2016.

[15] devttyS0. Embedded device hacking, 2017.

[16] L. R. Knudsen and M. J. Robshaw, "Brute Force Attacks," in *The Block Cipher Companion*, Information Security and Cryptography, pp. 95–108, Springer, Berlin, Heidelberg, 2011.

[17] K. Spett, *Cross-Site Scripting*, vol. 1, SPI Labs, 2005.

[18] J. Burns, *Cross site request forgery. An introduction to a common web application weakness, Information Security Partners*, 2005.

[19] M. Wu, R. C. Miller, and S. L. Garfinkel, "Do security toolbars actually prevent phishing attacks?" in *Proceedings of the CHI 2006: Conference on Human Factors in Computing Systems*, pp. 601–610, can, April 2006.

[20] N. Asokan, V. Niemi, and K. Nyberg, "Man-in-the-middle in tunnelled authentication protocols," in *Security protocols*, vol. 3364 of *Lecture Notes in Comput. Sci.*, pp. 28–48, Springer, Berlin, 2005.

[21] N. Gelernter, S. Kalma, B. Magnezi, and H. Porcilan, "The Password Reset MitM Attack," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy, SP 2017*, pp. 251–267, May 2017.

[22] D. Wang, J. Ming, T. Chen, X. Zhang, and C. Wang, "Cracking IoT Device User Account via Brute-force Attack to SMS Authentication Code," in *Proceedings of the the First Workshop*, pp. 57–60, Incheon, Republic of Korea, June 2018.

[23] Google. Android, the world's most popular mobile platform, 2012.

[24] R. Fielding, J. Gettys, J. Mogul et al., "Hypertext Transfer Protocol – HTTP/1.1," RFC Editor RFC2616, 1999.

[25] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)," *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 61–69, 2008.

[26] A. Warshavsky, A. Fiske, B. Cinarkaya, and R. Guest, "System, method and computer program product for performing one or more actions utilizing a uniform resource locator," *The US Patent*, vol. 8, pp. 990-144, 2015.

[27] S. G. Stubblebine and V. D. Gligor, "On message integrity in cryptographic protocols," in *Proceedings of the Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 85–104, May 1992.

[28] W. Chung-Ping and C-C. Jay Kuo, "Fast encryption methods for audiovisual data confidentiality," in *Proceedings of the In Multimedia Systems and Applications III*, vol. 4209, pp. 284–296, 2001.

[29] M. Aizatulin, A. D. Gordon, and J. Jan, "Extracting and verifying cryptographic models from C protocol code by symbolic execution," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS'11*, pp. 331–340, October 2011.

[30] M. Boreale, "Symbolic trace analysis of cryptographic protocols," in *Automata, languages and programming*, vol. 2076 of *Lecture Notes in Comput. Sci.*, pp. 667–681, Springer, Berlin, 2001.

[31] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *Engineering Secure Software and Systems*, vol. 6542 of *Lecture Notes in Computer Science*, pp. 58–72, Springer, Berlin, Germany, 2011.

[32] F. Gröbert, C. Willems, and T. Holz, "Automated Identification of Cryptographic Primitives in Binary Programs," in *Recent Advances in Intrusion Detection*, vol. 6961 of *Lecture Notes in Computer Science*, pp. 41–60, Springer, Berlin, Heidelberg, 2011.

[33] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42–51, 2002.

[34] C. R. Ramakrishnan and J. Rehof, "Attacking path explosion in constraint-based test generation," in *Proceedings of the nternational Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 366, 351 pages, Springer.

[35] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[36] I. You and K. Yim, "Malware obfuscation techniques: a brief survey," in *Proceedings of the 5th International Conference on Broadband Wireless Computing, Communication and Applications (BWCCA '10)*, IEEE, pp. 297–300, November 2010.

[37] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2016.

[38] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2014*, pp. 204–217, usa, June 2014.

[39] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC Editor RFC4627, 2006.

[40] T. Bray, J. Paoli, and Michael Sperberg-McQueen. C., "Extensible markup language (xml)," *World Wide Web Journal*, vol. 4, no. 2, pp. 27–66, 1997.

[41] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, pp. 209–220, ACM, February 2013.

[42] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2014.

[43] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering, WCRE 2003*, pp. 260–269, November 2003.

[44] R. Winsniewski, *Android–apktool: A tool for reverse engineering android apk files*, 2012.

[45] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed android applications," in *Proceedings of the European Symposium on Research in Computer Security*, pp. 293–311, 2015.

[46] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Co-located with ICSE 2011*, pp. 77–83, May 2011.

[47] S. Gunasekaran and V. Bargavi, "Survey on automation testing tools for mobile applications," *International Journal of Advanced Engineering Research and Science*, vol. 2, no. 11, pp. 2349–6495, 2015.

[48] Android Developers. Android debug bridge, 2014.

[49] E. H. Weigle, "High-speed and high-fidelity system and method for collecting network traffic," *The US Patent*, vol. 7, pp. 783-739, 2010.

[50] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal network protocol analyzer toolkit*, Elsevier, 2006.

[51] L. Richardson and S. Ruby, *RESTful web services*, O'Reilly Media, Inc., 2008.

[52] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," RFC Editor RFC1738, 1994.

[53] S. Ramanujam and M. Karuppiah, "Designing an algorithm with high avalanche effect," *IJCSNS International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 106–111, 2011.

[54] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 227–248, 1980.

[55] Xiaocong, Python wrapper of android uiautomator testing framework, 2014.

[56] mitmproxy. An interactive tls-capable intercepting http proxy, 2016.

[57] S. Parkes, *Sqlite: An embeddable sql database engine*, 2011.

[58] CNCERT/CC. China national vulnerability database, 2018.

[59] G. Tsirtsis and P. Srisuresh, "Network Address Translation - Protocol Translation (NAT-PT)," RFC Editor RFC2766, 2000.

[60] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the In USENIX Security Symposium*, pp. 95–110, 2014.

[61] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC 2010*, pp. 97–106, December 2010.

[62] A. Cui, M. Costello, and J. S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Proceedings of the NDSS*, 2013.

[63] H. Li, D. Tong, K. Huang, and X. Cheng, "FEMU: A firmware-based emulation framework for SoC verification," in *Proceedings of the 8th IEEE/ACM International Conference on Hardware/Software-Co-Design and System Synthesis, CODES+ISSS 2010*, pp. 257–266, usa, October 2010.

[64] Z. Wang, Y. Zhang, and Q. Liu, "RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing," *KSII Transactions on Internet and Information Systems*, vol. 7, no. 8, pp. 1989–2009, 2013.

[65] J. Chen, W. Diao, Q. Zhao et al., *Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing*, 2018.

[66] W. G. Halfond and A. Orso, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1 of *IEEE*, pp. 13–15, 2006.

[67] V. Philipp, F. Nentwich, N. Jovanovic, E. Kirda, K. Christopher, and V. Giovanni, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proceedings of the NDSS*, vol. 2007, p. 12, 2007.

[68] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and Communications Security, CCS'08*, pp. 75–87, usa, October 2008.

[69] D. Michael, C. Kozyrakis, and N. Zeldovich, *Nemesis: Preventing authentication & access control vulnerabilities in web applications*, 2009.

[70] G. Pellegrino and D. Balzarotti, "Toward Black-Box Detection of Logic Flaws in Web Applications," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2014.

[71] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online security analysis of cashier-as-a-service based web stores," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP 2011*, pp. 465–480, May 2011.

[72] Android Developers. Ui/application exerciser monkey, 2012.

[73] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE '13)*, pp. 224–234, ACM, August 2013.

[74] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2012*, November 2012.

[75] Z. Chaoshun and L. Zhiqiang, "Smartgen: Exposing server urls of mobile apps with selective symbolic execution," in *In Proceedings of the 26th International Conference on World Wide Web*, pp. 867–876, International World Wide Web Conferences Steering Committee, 2017.

[76] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, p. 1, 2012.

[77] C. Weidong, K. Jayanthkumar, and J. H. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *In USENIX Security Symposium*, p. 14, 1, 2007.

[78] B. Marshall, *The Protocol Informatics Project*, 2004.