Trampoline Over the Air: Breaking in IoT Devices Through MQTT Brokers

Huikai Xu[†1], Miao Yu[†1], Yanhao Wang[§], Yue Liu[‡,§],
Qinsheng Hou[*,§], Zhenbang Ma[§], Haixin Duan[†,§], Jianwei Zhuge[†2], and Baojun Liu[†]
[†]*Tsinghua University* [‡]*Southeast University* [*]*Shandong University*
[§]*Qi An Xin Technology Research Institute*

***Abstract*—MQTT is widely adopted by IoT devices because it allows for the most efficient data transfer over a variety of communication lines. The security of MQTT has received increasing attention in recent years, and several studies have demonstrated the configurations of many MQTT brokers are insecure. Adversaries are allowed to exploit vulnerable brokers and publish malicious messages to subscribers. However, little has been done to understanding the security issues on the device side when devices handle unauthorized MQTT messages. To fill this research gap, we propose a fuzzing framework named SHADOWFUZZER to find client-side vulnerabilities when processing incoming MQTT messages. To avoiding ethical issues, SHADOWFUZZER redirects traffic destined for the actual broker to a *shadow broker* under the control to monitor vulnerabilities. We select 15 IoT devices communicating with vulnerable brokers and leverage SHADOWFUZZER to find vulnerabilities when they parse MQTT messages. For these devices, SHADOWFUZZER reports 34 zero-day vulnerabilities in 11 devices. We evaluated the exploitability of these vulnerabilities and received a total of 44,000 USD bug bounty rewards. And 16 CVE/CNVD/CNNVD numbers have been assigned to us.**

1. Introduction

The rapid development of low-cost processors and wireless communication technologies motivates various manufacturers to apply Internet of Things (IoT) and Machine-to-Machine (M2M) [1] technology in their products. As one of the most commonly-used M2M communication protocols, MQTT (Message Queuing Telemetry Transport) is designed to make the most efficient data transmission over expensive and unreliable communication lines. As such, the protocol is defined with flexible interaction modes and simple message construction. Furthermore, instead of request/response, MQTT adopts the publish/subscribe (pub/sub) [2] model to transmit data between IoT devices (clients) and server applications (brokers). However, once security risks occur in the configuration of brokers, the advantages of MQTT have become a double-edged sword.

Previous Study on MQTT Protocol Security. The weaknesses in the authentication and authorization of MQTT protocol have resulted in some IoT devices and brokers being exposed to attackers. Andy et al. [3] summarized the implementation issues of MQTT, such as encryption and authentication. Similarly, some public reports [4], [5] demonstrated that more than 60% of anonymously accessible MQTT brokers leaked sensitive information. Even if a broker has authentication enabled, there is no guarantee that its credentials will not be stolen by attackers. Many manufacturers embed credentials or the sensitive information used to generate passwords into the firmware [6], [7], making them accessible to attackers. Some research [8]–[10] has demonstrated that such cryptographic misuse can be identified automatically. Because previous works do not explain how to use this sensitive information to carry out specific attacks, there has been little progress in understanding the security risks of hard-coded credentials. A recent work [11] also demonstrated that even popular IoT cloud platforms, such as AWS IoT, were vulnerable in managing sensitive data such as session and ClientId (i.e., client identifier). With these unreliable brokers, attackers can publish unauthorized messages to the devices (subscribers) that do not belong to them and manipulate devices remotely [12].

Research Gap. Previous works have shown that an attacker could send crafted messages to any target device via vulnerable brokers just by knowing the topic the device subscribes to. Therefore, it is critical to investigate the security of subscribers in the MQTT messages parsing phase. However, a few research has been done into security vulnerabilities that may arise when clients parse these crafted messages. Unlike attacking servers, typical client-oriented attacks necessitate the attacker taking complete control of the server; however, this is not necessary with MQTT. These insecure brokers (servers) can be considered trampolines for exploit payloads to be forwarded to subscribers. One conceivable but undiscussed situation is that when the client has a vulnerability while processing MQTT messages, an attacker can trigger the client's vulnerability by publishing unauthorized messages to the broker without taking complete control of the broker. Attackers can even execute arbitrary code on the device remotely without directly accessing it. We call this exploitation a *trampoline-over-the-air* attack. Considering that many brokers have authentication problems, it is crucial to find vulnerabilities in the processing of MQTT messages by IoT devices. Although many research works propose methods for fuzzing network protocols or IoT devices, they are inapplicable to detect the vulnerabilities described above.

Previous Study on Fuzzing Network Protocols. There are multiple works on fuzzing network protocols to find vulnerabilities.

- *Fuzzing on MQTT protocol.* Luis et al. [13] introduced five existing black-box fuzzing frameworks for

1. These authors contributed equally to this work.
2. Corresponding author. zhugejw@tsinghua.edu.cn

2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1614-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EUROSP53844.2022.00019

DOI 10.1109/EuroSP53844.2022.00019

MQTT [14]–[18]. These efforts look at how to discover vulnerabilities introduced by processing the header fields of the MQTT control packets. However, this type of vulnerability only occurs on the broker side and not on the subscriber side. Because the MQTT packet header can be controlled only if an attacker is a middleman or has taken over the broker completely. Some grey-box network protocol fuzzers, such as AFLNET [19] and MultiFuzz [20], can be utilized to find vulnerabilities on open-source MQTT implementations with customization. However, these tools require software simulation and instrumentation outside the actual device, which is not easy due to the complexity of the IoT hardware (e.g., vehicle's T-Box system).

- *Fuzzing on IoT Devices.* Generic black-box fuzzers like SNOOZE [21] and BOOFUZZ [22] support fuzzing on IoT devices. These tools require complex customization on message templates. There are also fuzzing approaches that require certain conditions for the device under test. For example, IOTFUZZER [23] utilizes the information carried by the IoT app to guide the fuzzing, and SNIPUZZ [24] relies on response information to optimize the mutation and requires the device to supply detailed response information. However, IoT devices using the MQTT protocol do not necessarily provide these conditions. What's more, all these black-box fuzzers focus on fuzzing the open services (e.g., HTTP) of IoT devices, and cannot be directly used to detect vulnerabilities in MQTT clients.

There are three challenges to detect vulnerabilities in IoT devices when clients process MQTT messages. **C1:** *Fuzzing on pub/sub model.* Traditional fuzzers apply to the C/S model but not to the pub/sub model, which MQTT adopts. MQTT messages are created in the publisher, transmitted via the broker, and handled by the subscriber. Rather than migrating the traditional methods to fit MQTT clients, we need an approach to leverage the pub/sub model feature to make the fuzzing system more streamlined and easy to implement. **C2:** *Ethical Consideration.* In the pub/sub model, the test cases (i.e., messages) sent from the fuzzer need to pass through the broker before being forwarded to devices. However, publishing a mass of fuzzing data to the broker over the air is illegal and dangerous for the vendors. **C3:** *Crash monitoring.* Traditional black-box fuzzers can monitor the crash by watching whether the target service is open or whether the network connection is broken. However, in the pub/sub model, the publisher and the subscriber are decoupled through the broker. The subscriber does not receive data from open service, and the fuzzer (always act as a publisher) cannot observe the connection status between the subscriber and the broker so that it cannot detect crashes.

Our Work. In this paper, we aim to explore client-side vulnerabilities that may cause trampoline-over-the-air attacks and evaluate their hazards. Specifically, we describe the threat model of the trampoline-over-the-air attack and create a fuzzing system named SHADOWFUZZER to find message parsing vulnerabilities on MQTT clients. SHADOWFUZZER uses a *shadow broker* instead of the original broker to monitor network connections, forward test cases, and circumvent ethical issues. Finally, we present details and exploits for these vulnerabilities case by case.

To tackle the above challenges, specifically, SHADOWFUZZER first leverages the pub/sub model of the MQTT protocol to perform fuzzing test. That is, the fuzzer is used as the publisher of the message, and the test cases are forwarded to the subscriber devices through the broker (C1). However, publishing test cases to the MQTT broker deployed in the cloud is dangerous and illegal. Hence we set up a shadow broker controlled by us to replace the original one. The shadow broker avoids the legal issue that affects the actual function of the broker in the real world (C2). Because controlling the shadow broker entirely, the fuzzer can monitor the TCP connection status between the shadow broker and the target device or waiting for other signals to judge whether a vulnerability is triggered (C3). We have released our code on GitHub [25].

We select 15 IoT devices communicating with vulnerable brokers to evaluate our fuzzing system. As a result, SHADOWFUZZER finds 34 zero-day vulnerabilities in 11 devices, including 22 command injection vulnerabilities, six buffer overflow bugs, one incorrect type conversion error, and five NULL pointer dereference bugs. At the time of paper writing, we have applied and been assigned 16 CVE/CNVD/CNNVD numbers. We assessed the exploitability of these trampoline-over-the-air vulnerabilities and totally received 44,000 USD bug bounty rewards. We also compare SHADOWFUZZER with other state-of-the-art fuzzers, and the results demonstrate that our system is more effective and targeted in finding vulnerabilities on MQTT clients. Overall, our paper makes the following contributions:

- We present a systematic study to discover vulnerabilities in IoT devices when handling unauthorized MQTT messages. Considering that more than half of the MQTT brokers on the Internet have severe authentication problems, IoT devices that communicate with these brokers will likely become the victim for bot masters once this exploitation is noticed. Therefore, it is essential to conduct this research for community to pay more attention on security issues of MQTT clients.
- To find such vulnerabilities, we propose a shadow broker to replace the original broker to implement a black-box fuzzing system called SHADOWFUZZER. We performed experiments on multiple real-world IoT devices, and in 11 devices, SHADOWFUZZER discovered 34 zero-day vulnerabilities, which can be used to launch trampoline-over-the-air attacks. Among these vulnerabilities, 17 can be exploited to launch such an attack from anywhere on the Internet and gain control of the target IoT devices. At the time of writing, we have not found this type of vulnerability in the CVE list.

2. Background

In this section, we provide an overview of the MQTT protocol and its security issues.

2.1. Overview of MQTT protocol

Machine-to-Machine Communication. M2M is an effective way to enable wireless and wired devices to connect and communicate directly with little or no human intervention. According to the communication paradigms,

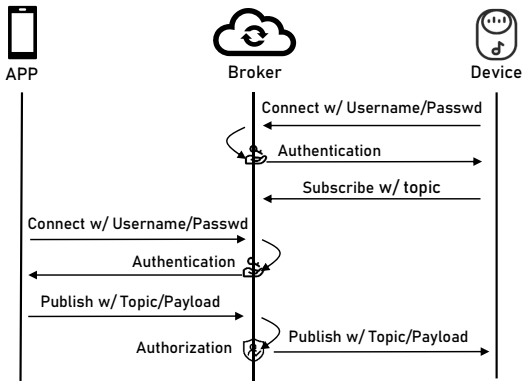

Figure 1: Pub/Sub mode of MQTT. The MQTT protocol works by exchanging a series of MQTT Control Packets in a defined way.

the M2M-based protocols can be divided into two different architectures: request/response model such as HTTP and CoAP, and publish/subscribe model such as MQTT. Regardless of the difference between the two communication models, they have in common that the authentication information (e.g., hard-coded credentials and seeds used to generate credentials) of clients is stored in the firmware of the devices instead of user's memory.

MQTT Protocol Model. MQTT is a lightweight protocol for transmitting data between devices over TCP protocol. Unlike protocols based on the request/response model, MQTT adopts a publish/subscribe architecture. Figure 1 shows the communication model of the MQTT protocol. The subscriber obtains messages from the broker (i.e., server) with a specific topic. The publisher transmits messages to the broker and identifies to whom the messages are forwarded through the topic. The broker is the center of the MQTT architecture. It is in charge of the data exchange between the publisher and the subscriber. During this process, the broker verifies clients' identities (i.e., publishers and subscribers) via the credentials sent from the clients, and the permissions of the clients also need to be checked when the clients publish or subscribe to messages.

The MQTT protocol works by exchanging a series of MQTT control packets, such as `CONNECT`, `PUBLISH`, `SUBSCRIBE`, and `PINGREQ`, which are used for establishing a connection, publishing messages, subscribing to messages, and session maintaining, respectively. There are three required fields in the `CONNECT` control packet for authentication: `ClientId`, `Username`, and `Password`. The `ClientId` is the unique identification of each client, which the broker uses to authorize the subscriber and publisher. And the pair of `Username` and `Password` can be used by the broker for both authentication and authorization.

2.2. Security Overview of MQTT

The MQTT protocol is designed with only a few security considerations, but its security issues in the real world mainly come from incomplete implementation and configuration.

Authentication. An MQTT client could communicate with a broker with or without authentication. Even though a client can prove itself with its certificate or `username/password`, there is still no guarantee that the identity will not be forged. Since MQTT is a M2M protocol, credentials are usually stored in the firmware of devices so that devices can connect to the broker actively.

There are two types of MQTT credentials in the IoT scenarios: shared credentials and exclusive credentials. One shared credential may be used for all devices with the same model, even the same brand. This type of credential is usually hard-coded in firmware images and applications. Attackers can easily extract them using unpacking and reverse engineering techniques. The exclusive credentials are more secure since they are always derived from the hard-coded seeds by some dynamic schemes such as OAuth [26] or token. However, hackers can still reproduce the generation process of the exclusive credential and access the target broker.

Authorization. Authorization is a mechanism to limit the permissions of clients. Most software in the broker-side of the MQTT protocol supports the Access Control List (ACL) for authorization. Brokers can use `username` or `ClientId` bound to ACL to determine which MQTT topic can be accessed by a client, such as publish or subscribe to a topic. However, if a broker allows anonymous access or uses a shared credential that is hard-coded in the devices to verify the client, it can only perform authorization on `ClientId`, which can be guessed or inferred by attackers according to prior work [11].

Secure MQTT Implementation. From a data security perspective, authorization is more critical than authentication. Because even if an attacker has bypassed authentication and connected to the broker, he cannot publish/subscribe to any messages if the broker authorizes topics strictly. Therefore, a secure MQTT broker must have sophisticated authentication and strict limits on the topics that can be accessed. For example, Message Queue for MQTT [27] is a lightweight messaging middleware provided by Alibaba Cloud for mobile Internet and IoT scenarios. It supports token-based authentication and authorization. When a client wants to publish or subscribe to messages, it first sends its identification and the topic it wants to access to an application server, which is in charge of client validation. If the client is valid and permitted to access the requested topic, the server asks the MQTT broker for a token with a privilege (R, W, RW) and expiry date and returns it to the client. The client can access the specified topic with the token to be the password.

Location of Broker. Typically, MQTT brokers are deployed in the cloud, where IoT devices and controllers communicate. But in some cases, brokers are deployed locally on the devices. For this kind of broker, the lack of cloud management makes it difficult to achieve secure authentication and authorization. Therefore, attackers can easily connect to the broker. A broker embedded in a device is typically used in LAN communication or inter-process communication. In the LAN communication scenario, the embedded broker can be used as a pivot to attack other subscribers in the same LAN (e.g., attack slave devices via the embedded broker of the master device in a Mesh network [28]). In the inter-process communication scenario, broker and subscriber are two separate processes in the same device. Although the broker process opens the MQTT service, the vulnerability may be triggered in the

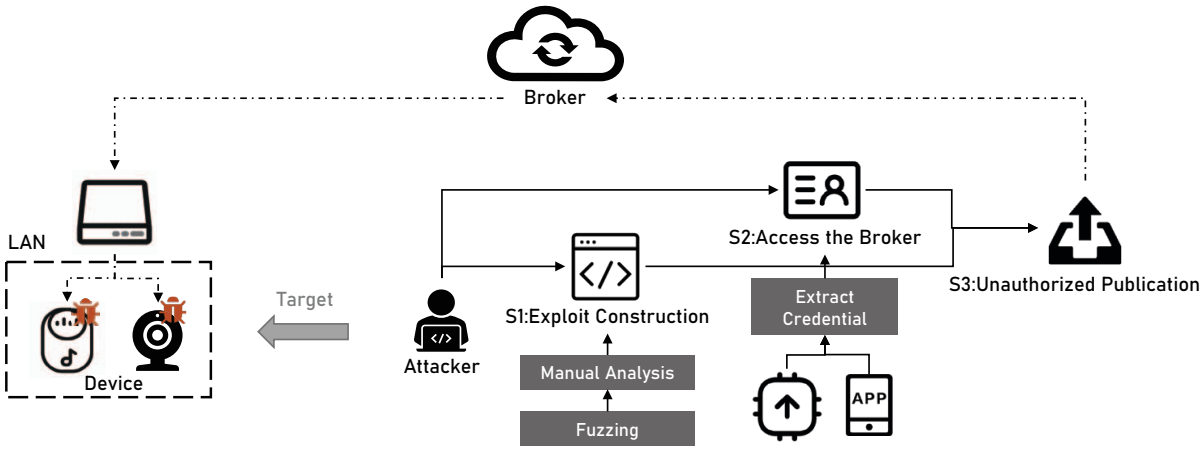

Figure 2: Overview of the trampoline-over-the-air attack.

subscriber process.

3. Threat Model and Attack Paradigm

In this section, we describe the threat model in detail and show how an adversary can launch a trampoline-over-the-air attack. Figure 2 presents the overview of the trampoline-over-the-air attack.

3.1. Threat Model

Motivation. The attack targets are the IoT devices communicating with the MQTT broker. The adversary aims to leverage the broker as a trampoline to transfer exploit messages to the target devices to trigger the vulnerabilities when processing the MQTT payload. Unlike previous IoT malware attacks such as Mirai [29], this attack has the following two characteristics.

- *High Penetrability.* Traditional attacks usually attack the open service of the device, requiring the adversary to access the device's service directly. However, in this threat model, devices can be compromised even if they cannot be accessed publicly (e.g., hidden under NAT). Conceivably, it's a terrible thing that an IoT camera in your home's LAN goes out of your control.
- *Good Stealthiness.* On the one hand, an adversary can subscribe to a wildcard like `"device/#"` to obtain information about all the online devices without performing a large-scale scanning to search devices. On the other hand, if the adversary utilizes the compromised devices to launch other attacks such as DDoS, it is difficult to trace the attack source because these devices may not be publicly accessible.

Assumptions. We assume that the adversary can connect to the MQTT broker successfully. Three possible flaws can lead to such an attack. (i) The MQTT broker does not require a password. (ii) The credentials are hard-coded in the mobile applications or the devices' firmware, which can be easily extracted. (iii) The credentials are derived from hard-coded and random information, which adversaries can replay with. We also assume that the broker is not configured correctly for authorization, making an authenticated adversary subscribe to any topic and publish messages to devices that do not belong to them. Typically, a broker with inadequate authentication also has an issue with authorization. It should be declared that an adversary has no control of the MQTT broker in this threat model.

Positions. As stated in §2.2, brokers in different locations may present different attack scenarios and also have different requirements for the position of the adversary.

- *Remote Broker.* A broker deployed in the cloud completely isolates publishers and subscribers, and both can be located anywhere they can access the Internet. Therefore, an adversary acts as a publisher, and there are no restrictions on the adversary's location in the threat model.
- *Local Broker.* A broker embedded in a device is typically used in LAN communication or inter-process communication. Since most IoT devices are located on a local area network, an adversary needs to be on the same LAN as the broker to access it in this scenario.

3.2. Attack Steps

Three steps are needed to launch a trampoline-over-the-air attack: construct the exploit, access the broker and publish unauthorized messages.

S1: Exploit Construction. An adversary manage to find vulnerability when the target device processes MQTT messages via vulnerability discovery techniques, such as fuzzing or manual analysis. Then they construct the exploit payloads as the MQTT messages and prepare for the real-world attack.

S2: Access the Broker. To make an MQTT broker trampoline the exploit, an adversary must connect to the broker successfully at the beginning. The adversary can directly send `CONNECT` control packet without credentials to connect to brokers that do not require authentication. For targets requiring hard-coded information to act as (or generate) credentials, an adversary needs to do reverse engineering to extract and infer the credentials.

S3: Unauthorized Publication. The topic of the exploit message usually contains the device's identity information, which can be obtained through a wildcard subscription. After gaining the necessary information, the

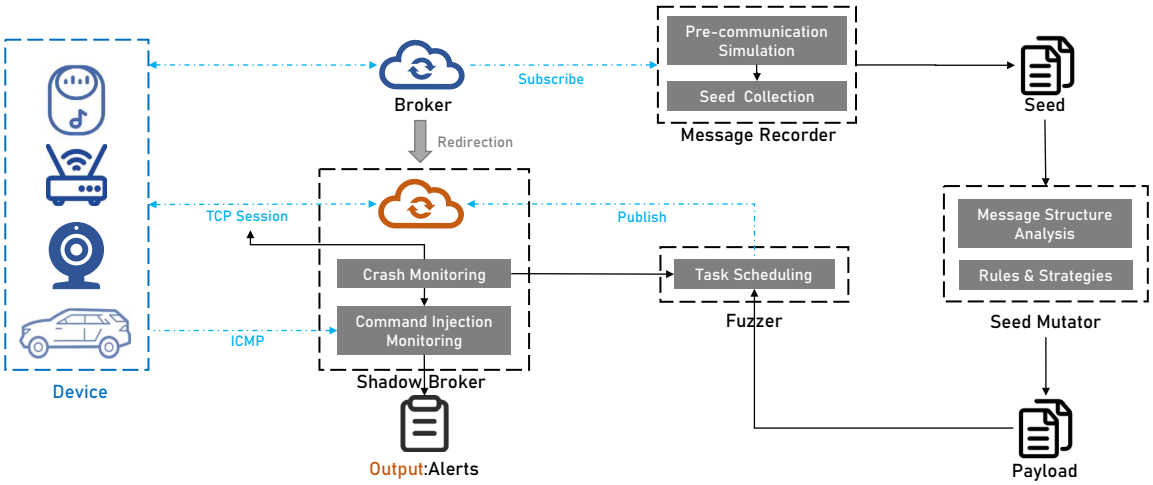

Figure 3: Overview of SHADOWFUZZER.

adversary publishes the exploit messages to the broker with the same topic subscribed by the target device. According to the MQTT topic, the broker then forwards the crafted messages to the subscriber and triggers the vulnerability.

4. Fuzzing System Design

In this section, we present the detailed design of SHADOWFUZZER. IoT devices receive messages from users as subscribers, so instead of opening ports to wait for data, they actively establish a long-lived connection to the broker to fetch messages continuously. Hence the idea is to have the device connect to a *shadow broker* that we control and utilize the broker to forward our test cases. To discover vulnerabilities in the processing of MQTT messages by the device, SHADOWFUZZER first simulates the device to collect initial messages from the actual broker. Next, it mutates the message based on its format and publishes the mutated test cases to the shadow broker. The shadow broker finally forwards the test cases to the device according to the topic. With the help of the shadow broker, SHADOWFUZZER can capture bug information once a vulnerability occurs while the device processes a test case.

SHADOWFUZZER consists of four modules: message recorder, shadow broker, seed mutator, and fuzzer. Figure 3 illustrates the workflow and the modules of SHADOWFUZZER. We will explain their workflow and introduce our solution to the problems in this section.

- *Shadow Broker* is the core of SHADOWFUZZER. This module has two capabilities. First, like all MQTT brokers, it is responsible for forwarding test cases published from the fuzzer to the tested IoT device. And it avoids the ethical issue of sending test cases to real-world brokers. Second, it assists the fuzzer in monitoring the network connection status to the tested device. (§4.1)
- *Message Recorder* simulates the operations of target clients (i.e., subscribers) to subscribe to the same topics and collects messages from the broker. These messages are published from publishers (i.e., controllers), such as mobile applications and web applications. We duplicate these messages and store them as initial seeds in our local database. (§4.2)
- *Seed Mutator* generates test cases for fuzzer based on initial seeds. It parses the message format and searches the corresponding format template to mutate each field of the message. According to related works [24], [30], random mutations are inefficient for generating valid structured data, such as MQTT messages. We also need to design effective mutation rules for triggering various kinds of vulnerabilities. (§4.3)
- *Fuzzer* acts as a publisher and sends mutational messages to the target device via the shadow broker. It is deployed on the same machine as the shadow broker to avoid unnecessary network delays. In this paper, we mainly focus on two types of vulnerabilities: memory corruption and command injection. How to discover these types of vulnerabilities while they occur in the subscribers? We will explain it in §4.4.

```c
int packet__read(struct mosquitto *mosq)
{
  /* If we've not got the MQTT control packet type, read one
       and save it.*/
  if (!mosq->in_packet.command){
    /* Read the first byte of the TCP payload, which is the
         MQTT control packet type. */
    read_length = net__read(mosq, &byte, 1);
    if (read_length == 1) {
      mosq->in_packet.command = byte;
    } else if (read_length == 0 ){
      /* If the read_length is zero, the shadow broker finds
           a crash. */
      log_print("Crash Found!");
      fuzz_notify_fuzzer(CRASH);
    }
  }
}
```

Listing 1: The C pseudocode to detect crash on the shadow broker.

4.1. Shadow Broker

Traditional black-box fuzzing tools mainly aim at open services (ports), unlike client-oriented fuzzing. Since the IoT device is the client in our threat model, we need to return the test cases to the device after the device connects to the broker. Because the actual broker is not under our

control, it is impossible to return data directly from the broker to the device. We take advantage of the MQTT protocol to perform fuzzing on the IoT clients, that is, using the fuzzer as the publisher and leveraging a shadow broker under our control to receive test cases published from the fuzzer and forward them to the IoT device.

Shadow broker is the core module of our fuzzing approach. It serves two purposes. First, the shadow broker replaces the actual broker to forward test cases published from the fuzzer to the device and avoids the ethical issue of sending malformed data to the actual broker. Second, the shadow broker monitors the TCP connection between the device and the broker to determine whether a crash occurs. No authentication and authorization is configured on the shadow broker so that any client can successfully build an MQTT connection with it. We implement the shadow broker based on Eclipse Mosquitto [31], one of the most popular open-source MQTT implementations written in C language.

To make the device to connect to the shadow broker, we need to redirect the device-to-cloud traffic to it. This part of the work requires manual assistance and will be introduced in §5.2. Because of the client-oriented fuzzing, it is impossible to monitor crashes by detecting whether the target port is open. However, controlling the shadow broker makes it feasible for us to detect crashes by monitoring the state of the TCP connection between the device and the shadow broker. Typically, the subscription session will not terminate until the program crashes or the device shutdown. While a crash occurs, the subscriber sends a TCP packet with an `RST` or `FIN` flag to the broker to terminate the TCP connection. This type of TCP packet does not contain any payload on the application layer, which is an abnormal behavior of the subscriber since each MQTT control packet must contain at least 2 bytes of the fixed header. Therefore, the shadow broker detects crashes by actively monitoring whether received TCP packets have application-layer payloads. Listing 1 is the pseudocode of detecting client-side crashes on the shadow broker. The shadow broker monitors the length of data read from the MQTT control packet's first byte each time. If the read length is zero, the shadow broker determines it has detected a crash.

4.2. Information Gathering

Pre-communication Simulation. For few of our targets, there is some necessary information to make the MQTT communication between the shadow broker and target device works, and we obtain it through pre-communication simulation. For example, in Figure 4, a cookie used in a subscriber's MQTT communication is created by the device during the preparatory phase before the MQTT communication. Its value is random and cannot be changed in any MQTT message sent from the broker to the device (until the subscription process restarts). Otherwise, the subscriber will not parse the rest of the message. Besides, as Figure 4 shows, a device gets the dynamic address and port of the broker from a server and uses them to communicate with the corresponding MQTT broker. We manually analyze the communication packets and MQTT messages to determine the critical information for MQTT communication. Then we simulate

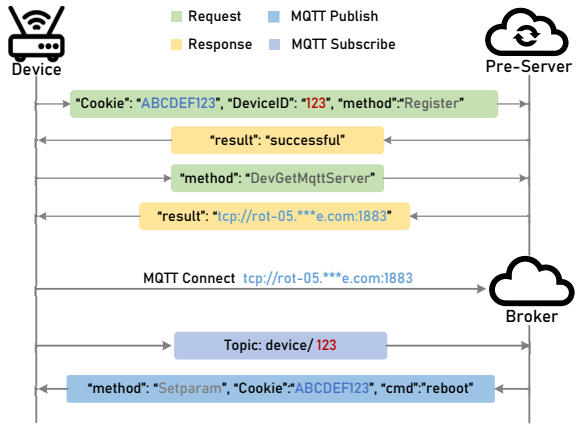

Figure 4: Pre-communication Sample.

```
domName = getDomainNameFromHttp();
char cmdbuf[128];
snprintf(cmdbuf, sizeof(cmdbuf), "ping %s", domName);
system(cmdbuf);
```

Listing 2: A command injection vulnerability example. This code fragment concatenates the domain name string received from the outside message to the "ping" command and calls the "system" function to run the Linux command.

the pre-communication process between different devices and their servers to obtain the information and use it in the following MQTT sessions.

Initial Seed Collection. Since we plan to find vulnerabilities in the device's processing of MQTT messages, we need to obtain the initial messages before fuzzing to generate mutated test cases. A man in the middle can monitor messages returned from the broker to the device. However, for the MQTT protocol, it can be achieved by simulating the device to subscribe to the same topic. Zhou et al. [6] created a phantom device (program) that mimics an actual device to assist the analysis. Similarly, we use the message recorder module to subscribe to the same topic as the tested device. The topic string usually contains unique information such as a device ID or a MAC address, and it can be obtained from the traffic or extracted from the firmware. For example, the function `mosquitto_subscribe` [32] in mosquitto C library is in charge of subscription, and the third parameter is the topic string, from which we can extract the topic. For real-world attackers, they can subscribe to a wildcard to get all online topics and then launch a large-scale attack.

After the message recorder subscribes to the topic, we trigger the device controller's UI controls (e.g., web app, mobile app) to generate various MQTT messages. The device controller acts as the publisher, it publishes these messages to the broker, and the broker forwards the messages to the message recorder. Finally, the message recorder stores these messages as initial seeds.

4.3. Mutation Rules

We design the following mutation rules to generate valid structured MQTT messages to trigger vulnerabilities.

Rules for Command Injection. A command injection vulnerability [33] is an attack that involves executing arbitrary commands on a system. It usually occurs while

passing unsafe user-supplied data to an interface, which has insufficient input validation. Listing 2 is an example of a command injection vulnerability. The code receives a domain name string from the HTTP payload and then concatenates the domain name to a format string to generate a command. The server process will run a valid `ping` command if a user inputs a valid domain name like `"www.google.com"`. However, if an attacker inputs a malicious string `"www.app$(reboot)le.com"`, the server process will first run the injected `"reboot"` command, which causes a DoS attack. In addition to the traditional rules such as command concatenation and command insertion used by other black-box fuzzers like Boofuzz, we also extract rules from GTFOBins [34] and use them to construct command injection strings for the string-type fields of the MQTT messages. GTFOBins is an open-source project listing a wide range of Unix binaries with details of leveraging them to execute system commands. For example, GTFOBins tells that arbitrary system commands can be executed by injecting malicious arguments into the `sed` command, a command-line utility for file searching, find and replace, insertion or deletion. A command like `"sed -n '1e reboot' /etc/hosts"` will cause the system to execute the `reboot` command. If the target program receives parameters from the outside and concatenates them to the `sed` command, a test case with the rule `"-n '1e reboot' /etc/hosts"` will make the device reboot. We select 20 items from GTFOBins and list them in Table 6.

Rules for Memory Corruption. Similar with IoTFuzzer, the mutator uses the following rules to detect memory corruptions. (i) Changing the size of strings. The mutator significantly increases the size of the string-type fields by appending plenty of redundant characters to trigger the buffer overflow bugs, or sets the fields to empty strings to find NULL pointer dereference vulnerabilities. Significantly, the rules also include generating a blank MQTT message and concatenating the original string with the malformed strings. (ii) Arithmetic operations or interesting values replacement. Similar to AFL and other fuzzers, the mutator changes the value of the integer-type and float-type fields with extremums, such as zero or negative numbers to trigger the integer overflow bugs.

Other Strategies. (i) For bool-type fields, the mutator changes their value from `True` to `False`, and vice versa. (ii) For list-type fields, the mutator adds elements or removes elements from the lists. (iii) The mutator will also change the type of various fields, such as replacing a string with an integer value or changing the list to a single value. These strategies also may lead to logic errors or crashes.

4.4. Vulnerability Monitoring

Specifically, ShadowFuzzer adopts different monitoring strategies for different types of vulnerabilities.

Crash Monitoring. As mentioned in §4.1, we leverage the shadow broker to monitor the status of the TCP connection between the subscriber and the shadow broker. Once the subscriber crashes, the TCP session will disconnect immediately, and the shadow broker would notify the fuzzer as soon as it notices this abnormal behavior. After receiving the notification, the fuzzer suspends the fuzzing process and record the current message as Proof-of-Concept.

Command Injection Monitoring. To monitor command injection vulnerabilities, previous work like Boofuzz detect command injection vulnerabilities by injecting system commands like `reboot` or `shutdown`, leading to the device shut down and thus blocks the fuzzing process. In our work, we leverage `ping` command to detect this type of vulnerability. The mutator inserts a command like `"ping -n 1 IP"` into the fuzzing payloads instead of the above commands, where `IP` is the IP address of the fuzzer. Once a command injection vulnerability is triggered, the device will send an ICMP Echo request packet to the fuzzer, which can be used as a signal that a command injection vulnerability exists. In this way, the fuzzer can detect command injection vulnerabilities without interrupting the fuzzing process. This approach is also used in [30].

5. Implementation

In this section, we describe how to look for IoT devices communicating with vulnerable brokers and present the implementation of ShadowFuzzer.

5.1. IoT Device Selection

To find targets that might be subject to the trampoline-over-the-air attack described in §3, we start with the following aspects to look for IoT devices communicating with the brokers configured with weak authentication.

Find Anonymously Accessible Brokers. There are multiple measures to find this kind of MQTT broker. For example, one can send MQTT `CONNECT` control packets with no credential to the broker and observe if the return code is zero, which means connecting successfully [35]. However, this approach carries legal risks. Several cyberspace search engines have integrated this feature. In our experiment, we take Censys [36] as the data source because this platform can provide the results of anonymously accessible brokers in both plaintext and TLS-wrapped MQTT protocols. We have requested and been approved for non-commercial research on Censys datasets. For Censys Search 1.0, rules `1883.mqtt.banner.connack.raw:0` and `8883.mqtt.banner.connack.raw:0` can list all the MQTT brokers opened on the default ports that allow anonymous access. It means that selecting the hosts using the MQTT protocol in port 1883 or 8883 and the return code in the `CONNACK` control packet is 0 (connection accepted). The rule has been changed in Censys Search 2.0 [37] after November 30, 2021. It is important to emphasize that we did not validate all the hosts in the result and only performed validation on the targets we list in this paper.

Find Brokers with Hard-coded Credentials Configured. Many manufacturers configure the same credential for all devices to facilitate management. Hard-coded credentials are typically stored in applications and firmware images as strings and passed as arguments to the functions used to set the credentials or establish an MQTT connection to the broker. Inspired by previous work done by Wang et al. [38], we leverage a backward taint analysis from a sensitive function to a constant string to find

```
1  /* Source Code: */
2  private ApiNetWorkHttpImpl() {
3    this.userName = "username";
4  }
5  this.options.setUserName(this.userName);
6
7  /* Smali Code: */
8  const-string v0, "username"
9  iput-object  v0, p0, ApiNetWorkHttpImpl->userName:String
10 iget-object  v3, p0, ApiNetWorkHttpImpl->userName:String
11 invoke-virtual MqttConnectOptions->setUserName(String)...,v3
```

Listing 3: Pseudocode of the backward taint analysis to find hard-coded credentials.

strings used as credentials. We built scripts to implement the backward taint analysis on mobile applications and firmware images. As Listing 3 shows, the scripts locate the caller function firstly (Line 5 & Line 11) to search the hard-coded username in an Android app (Line 3 & Line 8). This part of code calls a standard username setting function (i.e., `setUserName`) of a widely used MQTT SDK (i.e., Paho [39]). Because variable `v3` is used to store the argument related to the username, the scripts tracks the data flow from `v3` to its data source, the constant string `"username"`.

Find Brokers with Dynamically Generated Credentials Configured. We perform manual reverse analysis in IoT devices due to the difficulty of automatically detecting credentials generated dynamically. We try to reproduce the generation procedure and obtain temporary or permanent credentials. Although the credential generated by each device is different, attackers can still perform unauthorized subscription and publication after accessing the broker successfully if the broker lacks permission management.

After the selection, we extract the broker's address and purchase or rent the corresponding devices as our data set for fuzzing.

5.2. System Implementation

We implement our system with around 3,358 lines of Python code, 352 lines of C code (add to mosquitto), and 284 lines of Go code. This section will also introduce the manual efforts used in fuzzing processes.

Taint Analysis of Hard-coded Credentials. We implement the taint analysis of hard-coded credentials based on Janus [40], Androguard [41], and Binary Ninja Python API [42]. For mobile application analysis, we first use the Janus platform to search for the applications that use the Paho Android Service [39], a prevailing MQTT client library. Then for each application, the engine performs a backward taint analysis (implemented based on Androguard) to locate the hard-coded credential. For firmware images, we leverage Binary Ninja Python API to detect hard-coded credentials.

Fuzzing System. We build the shadow broker based on Eclipse Mosquitto version 1.6.8. We modify the source code of Mosquitto to add code to monitor TCP connections and communicate with the fuzzer. The IP addresses of the tested device and the fuzzer should be added into the broker's configuration file, and there is no need to configure credentials so that the shadow broker can receive any `CONNECT` request. The message recorder acts as a subscriber and the fuzzer acts as a publisher. Both are implemented based on Paho Python. We extract the valuable items in GTFOBins and import them into the mutation rules to find more command injection vulnerabilities.

Manual Efforts on Fuzzing. Since our test target is client and not server, the tested client needs to connect to our shadow broker before starting the test actively. Therefore, few manual efforts are inevitable for some of the test devices.

- *Certificate Replacement.* An increasing number of devices use TLS-based MQTT instead of the plaintext protocol for security consideration. The device first verifies the broker's certificate using the embedded CA certificate before establishing an MQTT connection with the broker. If directly redirect traffic to the shadow broker, the client can not build a TLS connection with the shadow broker because it fails to validate the broker's certificate with the original CA certificate. To solve this problem, we use the software `OpenSSL` [43] to create a self-signed root certificate to replace the original broker's CA certificate embedded in the device and use the new certificate to issue a server-side certificate. Finally, we place this server-side certificate into the shadow broker. In this way, while the device connects to the shadow broker, the shadow broker's certificate is validated with this self-signed root certificate.
- *Traffic Redirection.* In order to replace the original broker with the shadow broker, we need to redirect the traffic to the shadow broker. We choose different schemes depending on the location of the broker.
  - *Remote Broker.* There are multiple approaches to redirect the traffic sent from the test device to the shadow broker. Domain name hijacking is the most effective method. For the devices using a domain name to connect to the remote broker, we hijack the domain name at the gateway to the IP address of the shadow broker or modify the `hosts` file[3] in the operating system of the target device (if the subscriber contains this file). For the device only using an IP address to connect to the broker, we use `iptables` [44] in the gateway to redirect the traffic to the shadow broker.
  - *Local Broker.* We adopt two approaches to redirect the MQTT traffic for the devices embedding local broker. (i) For devices that load the broker address from a configuration file or command parameter, we change the IP address in the configuration file or parameter and reload the process. (ii) For the program that uses a hard-coded string (e.g., `localhost`, `127.0.0.1`) as the broker's address, we extract the binary executables and replace these strings with the IP address of the shadow broker so that the program can connect to it after been reloaded.

  Parts of these operations require that the device be available with a Linux shell via interfaces such as `Telnet` or `UART` (Universal Asynchronous Receiver/Transmitter) firstly. They are necessary because it is not easy to get TLS clients to change the direction of a network connection and establish it successfully. We adopt these engineering techniques on some devices to perform fuzzing on more devices and find more security issues.

3. Hosts file is used to maps hostnames to IP addresses.

5.3. Experiment Setup

Data Set. Using the methods introduced in §5.1, we finally found out and identified 5,179 MQTT brokers that could be accessed anonymously, and 28 were configured with hard-coded credentials. We also found two brokers were configured with dynamically generated credentials whose generation process could be deduced. From above vulnerable brokers, we select 15 of them to purchase or rent the corresponding IoT devices as our data set. Our choice is based on three principles: (i) the target devices should cover as many types as possible, (ii) the corresponding brands of devices should be as well-known as possible, (iii) the devices should be available to us and have a certain number of users. As can be seen in Table 4, there are multiple types of devices we selected, such as wireless routers, access points, smart speakers and automobiles. The brands of these devices also include well-known manufacturers like ZTE, Xiaomi, and Linksys. Among these IoT devices, 11 communicate with remote brokers, and 4 communicate with embedded brokers. All of these devices subscribe to MQTT messages from their brokers, respectively, and we only experiment on our own devices.

Testing Environment. We conduct our experiment on a CentOS 7 Server with Intel Core i9 8-core × 2.30 GHz CPU and 8GB RAM, and the devices get tested on the same local network.

Tools for Comparison. It is difficult to find tools suitable for fuzzing MQTT clients to compare with SHADOWFUZZER. We finally selected two open-source fuzzers for comparison: COTOPAXI [45] and BOOFUZZ configured with our shadow broker.

- COTOPAXI. Samsung produces this tool for security testing IoT devices using protocols such as HTTP, AMQP, and MQTT. The authors claim in the specification that the tool supports fuzzing MQTT clients. It provides a module named `client_proto_fuzzer` that allows users to test protocol clients, and MQTT is one of the supported protocols. The application will set up a server for listening requests from the tested client and return test cases in the same TCP connection.
- BOOFUZZ *with the shadow broker.* BOOFUZZ is a state-of-the-art fuzzer that has been widely used in recent years. Unlike COTOPAXI, BOOFUZZ requires a manually created template to guide the fuzzing process. Since this tool does not support client-oriented testing directly, our shadow broker is combined to assist experiments. We feed seed messages obtained from our message recorder to BOOFUZZ and create the template using BOOFUZZ API.

6. Evaluation

We evaluate our system on real-world devices to answer the following research questions:

- **RQ1**: Can SHADOWFUZZER find real-world vulnerabilities on the target devices (subscribers)? (§6.1)
- **RQ2**: Can these vulnerabilities be exploited in accordance with trampoline-over-the-air attack? What are the effects of these vulnerabilities? (§6.2, §6.3)

```
/* Seed Message 1 */
{"version":"3","contents":{"command":"support_cmd 01"}}
/* Seed Message 2 */
{"version":"3","contents":{"command":"tech_support_command 1"}}
/* Source Code */
int do_exec_command(){
  if (!strncmp(command, "tech_support_command", 0x14u) || !
       strncmp(command, "support_cmd", 0xBu)){
    sprintf(s, " &>%s", "/tmp/command_result");
    strcat(command, s);
    system(command)
  }
}
```

Listing 4: Two seed messages and the vulnerable code fragment of Alcatel-Lucent OAW-AP1101. Although different actions trigger these two seed messages, they are processed by the same piece of code.

6.1. Evaluation on Fuzzing

In this part, we present the vulnerabilities discovered by the SHADOWFUZZER and evaluate the efficiency and the accuracy to demonstrate that it is helpful to discover vulnerabilities under trampoline-over-the-air attack.

Real-world Vulnerabilities. As shown in Table 1, SHADOWFUZZER finds 34 zero-day vulnerabilities in 11 devices, and all of these vulnerabilities have been reported to the manufacturers once we verified them. At the time of paper writing, we have been assigned 16 CVE/CNVD/CNNVD numbers due to their severe security consequence and we exhibit them in Table 5.

Among them, 22 bugs are command injection vulnerabilities; six of them are buffer overflow bugs; one is incorrect type conversion error [46], and the other five are NULL pointer dereference bugs. All of these vulnerabilities can be triggered by publishing crafted payload to the vulnerable brokers from LAN or the Internet.

All the command injection vulnerabilities can be used to gain root privilege on devices. These vulnerabilities are easy to exploit and do not require attackers to do memory layouts. The five NULL pointer dereference vulnerabilities are triggered by a blank message. There is no limit on the minimum length of messages in the MQTT protocol specification. However, in some MQTT client libraries like Eclipse Mosquitto, the processing of zero-length messages is dangerous. When receiving an MQTT message with no content, Mosquitto sets the pointer of the message payload to NULL. If developers are not aware of this issue, they will likely use this NULL pointer to parse the data directly, which leads to a NULL pointer dereference error.

Efficiency. We measure the efficiency of the fuzzing system by the number of vulnerabilities found (command injection and memory corruption, respectively), the total time it took to execute all test cases, and the number of messages generated. The results are listed in Table 2.

We experimented on COTOPAXI v.1.7.0 (the newest version before the submission) and launched the `client_proto_fuzzer` with the MQTT protocol. Accordingly, we redirected the traffic from the original broker to the COTOPAXI server and managed to get the client to connect to it. However, each time a device tried to send a `CONNECT` control packet to connect to the server, COTOPAXI only returned a raw TCP packet with random payload, not the `CONNACK` packet expected. It prevents the client from successfully establishing an MQTT connection with COTOPAXI, and thus it is impossible to perform a fuzzing test on the MQTT message parsing stage. We

TABLE 1: Vulnerabilities. For the bug type, NPD means NULL pointer dereference, CI represents command injection, BO means buffer overflow, TE means incorrect type conversion. For the bug effect, RCE represents remote code execution, DoS represents denial of service. # represents the number of vulnerabilities. S-cred represents shared credential. D-cred represents dynamic credential.

Manufacturer	Type	Model	Vulnerability			Broker		Field
			Type	#	Effect	Location	Authentication	
ZTE	Wireless Router	E8820	NPD	1	DoS	Remote	S-cred	Home
TOPSEC	Access Point	TAP-62200	NPD	1	DoS	Remote	✗	Office
			CI	3	RCE			
			BO	1	RCE, DoS			
Alcatel-Lucent	Access Point	OAW-AP1101	NPD	1	DoS	Remote	✗	Office
			CI	3	RCE			
			BO	1	RCE, DoS			
HAN Networks	Access Point	AP211	NPD	1	DoS	Remote	✗	Office
			CI	3	RCE			
			BO	1	RCE, DoS			
Cetron	Access Controller	G500SE	CI	3	RCE	Remote	S-cred	Office
Xiaomi	Smart Speaker	Xiao AI Speaker Pro	CI	1	RCE	Local	✗	Home
Totolink	Wireless Router	T10	CI	6	RCE	Local	✗	Home
			TE	1	DoS			
			BO	1	RCE, DoS			
SAIC-GM-Wuling	Vehicle	New Baojun	CI	1	RCE	Local	✗	Transportation
			BO	1	RCE, DoS			
Rokid	Smart Speaker	Rokid Mini	CI	1	RCE	Remote	D-cred	Home
Linksys	Wireless Router	Velop	CI	1	RCE	Local	✗	Home
			NPD	1	DoS			
Brilliance Auto	Vehicle	V7	BO	1	RCE, DoS	Remote	✗	Transportation

analyzed the source code of COTOPAXI and found that the module does not set up the server according to the specific application layer protocol, but only listens on a TCP or UDP port, waits for requests, and returns random data. COTOPAXI does not follow the protocol state machine to establish a full MQTT connection. As a result, we did not compare our approach with this tool.

With the help of the shadow broker, BOOFUZZ has the ability to send test cases to the shadow broker and eventually to the IoT devices. We made templates for each seed of each device and loaded BOOFUZZ with the shadow broker to test them. For each device, BOOFUZZ only ran for no more than 24 hours. From Table 2, we can see that SHADOWFUZZER generated fewer test cases and took less time but discovered more vulnerabilities than BOOFUZZ. This is because (i) to support more protocols (e.g., HTTP), boofuzz has plenty of items in its mutation rules that are useless for the MQTT protocol, but SHADOWFUZZER are more targeted to the MQTT protocol and IoT devices. (ii) SHADOWFUZZER contains mutation rules that BOOFUZZ does not support. For instance, the command injection vulnerability of Linksys Velop wireless router was triggered by the test case generated by the mutation rule extracted from GTFOBins, which we have introduced in §4.3. Furthermore, all the NULL pointer dereference vulnerabilities discovered by SHADOWFUZZER were triggered by an MQTT message with no payload. However, BOOFUZZ mutates each field based on templates, not generating such an empty message. (iii) To detect command injection vulnerabilities, BOOFUZZ inserts `reboot` command into test cases, making devices reload multiple times during the fuzzing. As the improvement, SHADOWFUZZER inserts `ping` command and listens for ICMP packets to detect command injection bugs, which avoids interrupting the fuzzing process.

We do not adopt the approach like COTOPAXI to send the messages from server to client directly. Although it is a general idea to perform client-oriented fuzzing, it is not necessary for a third-party communication protocol such as MQTT. Because we can directly leverage the feature of the MQTT protocol to publish test cases from publisher to subscriber, which is easy to implement with the shadow broker and open-source MQTT SDKs.

False Positive. As Table 3 shows, there are several false positives in the crash and command injection alerts monitored by SHADOWFUZZER. Two reasons lead to the false positives in command injection vulnerabilities. On the one hand, several bugs occurred in the same code fragment. We manually analyzed all the bug alerts and excluded the duplicate alerts. As shown in Listing 4, because both seed messages result in executing to the vulnerable branch (Line 7), test cases generated based on either can trigger the vulnerability in line 10. On the other hand, one command injection vulnerability returns multiple ICMP echo request packets and makes SHADOWFUZZER report several times. For Totolink T10 wireless router, the `ping` command injected in the preceding message is executed by the vulnerable program more than once, causing the shadow broker to receive multiple ICMP packets. The false positives of crash are caused by unknown network problems. As can be seen from the table, parts of the devices have poor handling of network data, resulting in frequent disconnection of the network session during fuzzing.

6.2. Practical Implications Evaluation

Global Distribution of the Vulnerable Brokers. The authentication and authorization issues on MQTT brokers are the root cause of the trampoline-over-the-air attack. There are `124,586`[4] MQTT brokers globally reported by Censys. Of all these brokers, `74,500` (60%) can be accessed anonymously, which can be used to lauch the trampoline-over-the-air attack. We draw a figure to display the global distribution of these brokers. The result is shown in Figure 5. The top three countries are South Korea, China, and the United States, containing `27,413`, `20,220`, and `6,417` vulnerable brokers, respectively. Compared to the result in 2018 [5], the number of MQTT brokers has tripled. However, the proportion of the anonymously accessible brokers only decreased from 65% to 60%, indicating that the security of MQTT brokers remains a low priority.

4. The data was obtained on October 15, 2020.

TABLE 2: Fuzzing Efficiency. `Total` represents the total number of reported bugs in the five fuzzing rounds, `Time` means the average time of the five rounds, `Messages` illustrates the number of test cases.

Device	ShadowFuzzer						BooFuzz w/ Shadow Broker					
	#Total	#CI	#Crash	Time	#Messages	#Seed	#Total	#CI	#Crash	Time	#Messages	#Seed
ZTE E8820	1	0	1	1.7h	6049	17	0	0	0	>24h	429548	17
TOPSEC TAP-62200	5	3	2	1.42h	5002	18	2	1	1	>24h	326788	18
Alcatel-Lucent OAW-AP1101	5	3	2	1.4h	5002	18	3	2	1	>24h	326788	18
HAN-Networks AP211	5	3	2	1.45h	5002	18	2	1	1	>24h	326788	18
Cetron G500SE	3	3	0	1h	3473	16	1	1	0	>24h	231095	16
Rokid Mini	1	1	0	0.2h	694	9	1	1	0	6.9h	23475	9
Xiaomi Xiao AI Pro	1	1	0	0.1h	351	3	0	0	0	6.1h	20758	3
Totolink T10	8	6	2	0.39h	1177	16	3	2	1	>24h	84124	16
SAIC-GM-Wuling New Baojun	2	1	1	0.09h	252	5	2	1	1	4.5h	15684	5
Linksys Velop	2	1	1	1.1h	3491	13	0	0	0	>24h	248867	13
Brilliance Auto V7	1	0	1	0.08h	217	1	1	0	1	4.3h	13720	1

TABLE 3: Fuzzing Accuracy. `Total` represents the total number of the reported bugs, `ACC` is the accuracy and obtained by a division operation (confirmed/reported).

Device	#Reported			Confirmed			ACC
	#Total	#CI	#Crash	#Total	#CI	#Crash	
E8820	1	0	1	1	0	1	100%
TAP-62200	8	5	3	5	3	2	62.5%
OAW-AP1101	6	4	2	5	3	2	83.3%
AP211	7	5	2	5	3	2	71.4%
G500SE	4	4	0	3	3	0	75%
Rokid Mini	1	1	0	1	1	0	100%
Xiao AI Pro	1	1	0	1	1	0	100%
T10	13	8	5	8	6	2	61.5%
New Baojun	2	1	1	2	1	1	100%
Linksys Velop	2	1	1	2	1	1	100%
V7	1	0	1	1	0	1	100%

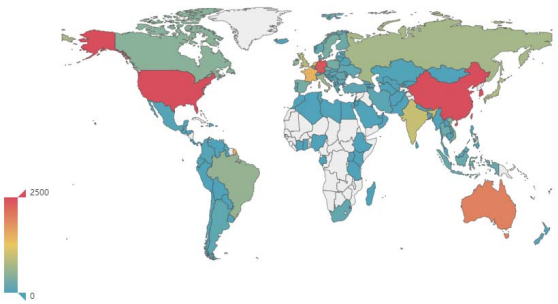

Figure 5: Global Distribution of the Weak-authenticated Brokers. The top three countries are South Korea, China and the United States, which contain `27,413`, `20,220` and `6,417` vulnerable brokers respectively.

Exploitability and Harmfulness of the Vulnerabilities. As Table 1 shows, the vulnerabilities we found affect many aspects of people's lives, including home life, office environment, and transportation. These vulnerabilities can be exploited in a different way than traditional attacks. Previous malware like Mirai focuses on the open services (ports) of a device, requiring an attacker to know the IP address and port of the device. However, in a trampoline-over-the-air attack, attackers just need to know the topic to which the device subscribes after finding an exploitable broker. The topic usually includes the device's ID, such as the `VIN` of a vehicle, which can be guessed or obtained by unauthorized subscription. Attackers can exploit vulnerabilities that lead to remote code execution to publish malicious messages to all online devices for indiscriminate attacks and construct a botnet as Mirai did.

To prove the trampoline-over-the-air attack is exploitable and harmful in reality, we built automated exploit scripts to demonstrate the exploitability of these vulnerabilities to third-party vulnerability assessment agencies [47]. It is important to note that all the IoT devices under test belong to us. For example, we exploited the command injection vulnerability of the New Baojun vehicle to gain the root privilege of the vehicle's T-Box unit and then played with the CAN bus to perform some hazardous actions such as opening the door and igniting the engine. In another instance, we took advantage of the command injection vulnerability of the Alcatel-Lucent wireless router to take over the device entirely from the Internet. We also exploited the buffer overflow vulnerability of Totolink wireless router successfully. There is no length limitation in the vulnerable message, and a stack overflow occurs when the length of the `url` field exceeds 84 bytes. We injected shellcode into a fixed heap address and tampered with the return address with shellcode's address, hijacked the program control flow, and got the device shell. We received `44,000` USD bug bounty rewards and helped these manufacturers fix the vulnerabilities we discovered.

6.3. Case Study

We present four cases to demonstrate the effectiveness of ShadowFuzzer and prove that the trampoline-over-the-air attack does exist in reality. The remaining cases are shown in Appendix §C. Among the 11 devices, ZTE E8820, TOPSEC TAP-62200, Alcatel-Lucent AP1101, HAN AP211, Cetron G500SE, Rokid Mini, and Brilliance V7 communicate with remote brokers, and others communicate with local brokers. The brokers of ZTE E8820 and Cetron G500SE are configured with shared credentials, while Rokid Mini's broker is configured with dynamic-generated credentials. The brokers of other devices are anonymously accessible.

OAW-AP1101/TAP-62200/AP211 Access Point. We found five severe vulnerabilities in Alcatel-Lucent OAW-series access point devices, such as OAW-AP1101. Administrators can use the Cloud Services Platform (CSP) to control these devices remotely over the MQTT protocol. Although there are hard-coded credentials (i.e., username and password) in the devices' firmware, the broker does not verify them and lacks authorization management, allowing attackers to exploit the vulnerabilities to execute arbitrary commands on all devices connected to the CSP. These access points have been deployed in various sites such as hospitals and office environments, providing at-

```
{"messageID":"e98b1234-1234-4321-xxxx-xxxxxxxxxxxx",
 "method":"basicinfo",
 "macAddress":"11:22:33:FF:FF:FF",
 ...}
/* The code of processing the above message in the
     executable file. */
char dest[88];
...
v8 = (const char *)json_object_get_string(v63);
strcpy(g_method, v8);
...
if ( strcmp(g_method, "setSsh") )
{
    memset(dest, 0, 0x32u);
    v59 = strlen(g_method);
    memcpy(dest, g_method, v59 + 1);    // stack overflow
    printf("methodstr=%s\n", dest);
}
```

Listing 5: A vulnerable message of Alcatel-Lucent access point and the processing code.

```
{"uuid":"AABB-1234","type":"status",
 "TS":"2021-03-09T13:22:34Z",
 "data":
 {"ap_bssid":"AA:BB:CC:DD:EE:FF","band":"5G",
  "sta_bssid":"AA:BB:CC:DD:EE:FF",
  "rssi":"", "interface":"ath1"}}
/* The code of processing the above message in the shell
     script. */
BSSID = "$(jsonparse data.sta_bssid < $PAYLOAD_PATH)"
TODIE = "$(find ${TESS_CS_SURVEYS} ${TESS_CS_PENDING_STEERS}
      - iname ${BSSID} | grep -v '.OLD')"
```

Listing 6: A vulnerable message of Linksys Velop wireless router and the processing code.

tackers access entries to break into the internal networks. After further study, we found the same vulnerabilities also exist in other manufacturers' access point devices such as TOPSEC TAP-series APs and HAN Networks HAN-series APs. Considering the domain names of these CSPs are "`*.han-networks.com`", we speculated that HAN Networks is the Original Equipment Manufacturer (OEM).

Listing 5 is one of the vulnerable messages and the code to process it. The mutator mutates the `method` field of the JSON-structured data and changes the value to a long string. While processing the message, the program uses the `memcpy()` function to copy the value of the `method` field to a stack-based buffer, which is only 88 bytes long. However, the third parameter, i.e., the length to copy, is the length of the source buffer. When the length of the `method` field is too long to exceed the size of the destination buffer significantly, a stack buffer overflow occurs. Attackers can leverage the vulnerability to construct a crafted exploit, then publish the malicious message with the specified topic to the remote broker located in the CSP so that the broker forwards the message to the specific victim device.

Linksys Velop Wireless Router. Linksys Velop [48] is a series of wireless router that supports Mesh networking between multiple devices in the same LAN. A `mosquitto` broker without authentication is embedded in Linksys Velop wireless router for device-to-device interaction. A command injection vulnerability in the process of message can be triggered by injecting a malicious argument "`-exec command`" to a `find` command to execute an arbitrary system command. As shown in Listing 6, the value of the "`sta_bssid`" field is extracted from the message and is finally passed to the `find` command as an argument. Therefore, attackers in the same LAN can inject "`-exec command`" into the "`BSSID`" variable to make the device run the malicious command. This vulnerability was discovered by a GTFOBins rule.

Rokid Mini Smart Speaker. One command injection vulnerability has been found in the Rokid Mini smart speaker. The problem occurs on the `runShell` action of the MQTT message, and attackers can execute arbitrary commands by injecting malicious commands into the `cmd` field. What is different from the above cases is that the MQTT credential of the device is exclusive and generated dynamically, which allows the broker to separate permissions with different credentials. Because the MQTT protocol is M2M-based, attackers can deduce the generation process of the dynamic credential, reproduce and use it to connect to the broker, and then publish malicious messages to the device they dominate. After further research, unfortunately, we notice that the authorization of this broker is also insufficient, allowing attackers to publish messages to arbitrary devices from anywhere on the Internet.

ZTE E8820 Wireless Router. The users can use a mobile app to manage this wireless router. The protocol used between the app and the router is MQTT, and the broker authenticates the clients by a shared client certificate, which is hard-coded in the firmware images. After receiving a new message sent from the broker, the device calls `strstr(payload,"method")` to determine whether a "`method`" string is included in the payload of the message. If the payload length is zero, the pointer of the payload (the first parameter of strstr) is null, triggering a NULL-pointer-dereference crash. Attackers can use this bug to attack the MQTT service of all the online devices with this vulnerability. It is noteworthy that this wireless router first accesses a server to obtain a cookie value and the address of the MQTT broker used for the later communication. We used the scheme mentioned in §4.2 to simulate this process and returned the cookie and the shadow broker's address to the devices to make the device connect to our shadow broker successfully.

7. Discussion

In this section, we discuss the root causes of the security issues in MQTT-based communication, the limitations, the ethical considerations, and the bug fixes of our work.

7.1. Root Cause Analysis

Flaw of M2M. M2M is a widely-used communication model in IoT scenario. To simplify the communication between the devices and the cloud servers, some manufacturers store the credentials in the devices' firmware images, which helps the devices connect to servers actively. Though some devices use a dynamic generation approach to obtain a provisional credential from the cloud, the generation process can still be deduced. For instance, Rokid Mini Smart Speaker sends a set of values, such as `sn`, `deviceId` and `secret`, to the cloud and receives a dynamically generated password to connect to the broker. However, these values are still fixed in one device, which can be extracted from the device's firmware. Therefore, currently, authentication problems are still common in the products of various manufacturers.

Unreliable Server. For protocols based on request/response architecture like HTTP, the client's real identity is unknown to the server. Therefore there are various

authentication and protection mechanisms on the server-side. In contrast, MQTT is a loosely coupled protocol based on pub/sub architecture, and the message receiver (i.e., the subscriber) is the role of client. From client's perspective, the server is entirely trustworthy regardless of the middleman on the connection, making the subscriber rarely check the validity of the received messages. However, if the broker is set up with weak authentication and authorization, an unauthorized publisher can send malicious messages to the target device. As a consequence, subscribers have to face various security risks even if they are only clients.

Misuse of MQTT Protocol.. Due to the simplicity and versatility of MQTT, there exists some misuses of this protocol. The pub/sub model of MQTT is similar to the bus communication system, it can not only be used for device communication, but also for inter-process communication. The Totolink T10 wireless router, for instance, opens a naked MQTT service and uses MQTT as an inter-process communication protocol to transmit HTTP requests between processes. However, it introduces a new attack surface that attackers can publish malicious HTTP data to the subscriber process from outside without HTTP authentication. On the other hand, some cases support transmitting system commands to the devices over the MQTT protocol. It is a dangerous action which attackers can use to inject malicious commands into the messages. We regard these behaviors as misuses of the MQTT protocol.

7.2. Mitigation

Sophisticated Authentication. Developers are still necessary to adopt sophisticated authentication measures based on the following considerations. On the one hand, it can prevent attackers from easily obtaining credentials. On the other hand, sophisticated authentication makes it convenient for the broker to isolate user privileges. For example, after binding with the app's user, the device can use the information of the user's account as a secret to generating a credential because this information cannot be hard-coded into the firmware.

Tightly Coupled Authorization of Device and User. In fact, the problems involved in trampoline-over-the-air attacks are mainly caused by lax permissions isolation, which provides conditions for unauthorized attackers to launch peer-to-peer attacks. Administrators of the broker should bind the user identifier with the device-side `ClientId` or `Username` once the user adds the device to his account on the application. In this way, the attacker who uses a different `ClientId` or `Username` can not publish messages to this device because of the tightly coupled authorization. Although this measure has been adopted by most public IoT cloud platforms, many manufacturers still ignore this issue in their private brokers.

Attack Detection on the Broker. Typically, the broker is only in charge of forwarding the messages without modifying or checking the message content, allowing the attack payload to be passed to subscribers without any constraints. One solution is to add a message content filtering module like WAF (Web Application Firewall) [49] to detect possible exploits in messages published from attackers.

7.3. Limitations

Device Type. With the help of the shadow broker, our fuzzing framework can be applied to any device that uses the plaintext MQTT protocol, regardless of the operating system type. However, certificate replacement is necessary for fuzzing clients communicating with brokers on TLS-based MQTT. It is necessary that the device can provide a Linux shell for certificate replacement. Hence the target device needs to be Linux-based, such as OpenWrt [50], one of the prevailing embedded device systems. Although this method increases few conditions for black-box fuzzing, it is an effective and common engineering technique for fuzzing on clients.

Device Selection. Successful exploitation of these vulnerabilities requires a combination of vulnerable brokers. Devices that communicate with brokers in the full-fledged IoT cloud platform such as Google IoT Cloud, Amazon AWS, Microsoft Azure, and Samsung SmartThings are less vulnerable to the trampoline-over-the-air attack because the authentication of these MQTT brokers is generally robust enough. Our fuzzing approach can be applied to these devices; however, even if vulnerabilities are found, attackers will not be able to exploit them because of the robust authentication mechanisms in the cloud. However, as described in §6.2, there are still many MQTT brokers with authentication problems around the world. Individuals or companies may set up these brokers to communicate with their devices, and the security of these brokers and devices is equally worthy of attention.

7.4. Ethical Considerations

In order to avoid ethical violations, we mainly consider the following four aspects. (i) To find brokers on the Internet that allow anonymous access, we extracted information from the Censys database rather than establishing MQTT connections directly to real-world brokers. It is necessary to indicate that we did not perform a large-scale validation after obtaining Censys's results and only performed validation on the targets mentioned in this paper. (ii) To implement a fuzzing test on MQTT clients, we used a shadow broker instead of the original broker to avoid publishing test cases to the actual broker. (iii) We only experimented on the IoT devices belong to us and did not send messages to other devices. (iv) We have followed the best security practices and reported all the vulnerabilities to manufacturers as soon as we found them. After the notifications, we left enough time for the manufacturers to fix the vulnerabilities, and this is the best we can do from a security researcher's point of view.

7.5. Bug Fixes

After confirming the vulnerabilities, the manufacturers of the tested devices did not disclose the details of the fix to us. As a result, we re-examined the vulnerabilities six to twelve months after reporting them.

Nine manufacturers have already fixed the vulnerabilities. Concretely, HAN Networks/TOPSEC/Alcatel-Lucent have fixed the client-side vulnerabilities and configured a hard-coded credential on the broker. ZTE has confirmed both the broker-side and device-side vulnerabilities. We

can not use the hard-coded certificate to connect to the broker now, and the NULL pointer dereference bug has been fixed in the firmware version `2.0.14`, according to the statement [51]. Xiaomi has fixed the command injection bug, and the embedded MQTT service is not accessible from LAN currently. Totolink has repaired the reported vulnerabilities in the firmware version `4.1.8`. Linksys added an ACL file named `strict.acl` in the embedded `mosquitto` broker with strict restrictions on readable/writable topics. The broker of Brilliance Auto V7 has been closed, so we can not connect to it. SAIC-GM-Wuling verified and confirmed the vulnerabilities as soon as we reported to them. This manufacturer has repaired the client-side and broker-side vulnerabilities on the New Baojun vehicle. Cetron and Rokid have received and confirmed the vulnerabilities but have yet to complete a fix.

8. Related Work

We have introduced the most related works and compared them with our work in previous sections. In this section, we discuss other works related to MQTT security and vulnerability discovery on IoT devices.

8.1. MQTT Protocol Security.

Implementation Issues. Previous works [4], [12], [52]–[54] on MQTT security mainly focus on the issues of broker's authentication and authorization. Lucas Lundgren [52] proposed that MQTT brokers without authentication could be used to control botnets to evade investigation. Firdous et al. [55] summarized the MQTT protocol's attack surfaces and possible threats, like sensitive data leakage and unauthorized publication. Unlike the above works, we focus on the MQTT security issues on the device side and propose a comprehensive approach to detect vulnerabilities of parsing MQTT messages.

Reinforcement. Due to the flaws in the MQTT security mechanism, some studies [56]–[60] proposed approaches to strengthen security. For example, Niruntasukrat et al. [58] introduced a new authorization mechanism based on OAuth. Singh et al. [59] designed multiple schemes to solve the difficulty of certificate management. Bali et al. [60] used the chaotic algorithm to implement a lightweight authentication mechanism for MQTT. Although these measures attempt to strengthen the protocol in many ways, they bring more complexity for resource-constrained IoT devices.

8.2. Vulnerability Discovery on IoT Device.

With the increase in IoT devices and related attack risks, researchers [61] pay more attention to the methods for discovering the vulnerabilities of IoT devices, which mainly includes two types of solutions: static and dynamic analysis methods.

Static Analysis. Most static analysis methods focus on specific vulnerability issues, such as weak keys [62], authentication bypass [63], memory bugs [64] and taint-style vulnerability [65]. KARONTE [66] focuses on finding vulnerabilities cross binaries via tracking the data flow with the label of inter-process communication. However, static methods may result in a significant amount of false positives and are limited to the problem of how to obtain the device's firmware.

Dynamic Analysis. In contrast, dynamic methods are more suitable for discovering vulnerabilities in the black-box IoT devices. Muench et al. [67] implemented a system based on Avatar [68] and Panda [69] to analyze the universality of traditional anomaly state detection methods for the IoT device. In comparison, SHADOWFUZZER can generate fuzzing data without complex data flow analysis. Wang et al. [30] discovered vulnerabilities on device's web management interface based on a lightweight black-box fuzzing method, which could only find vulnerabilities in web services. All these black-box fuzzers are mainly for the device's open services, like web services, or other services running on an open port, which can not be applied to MQTT clients. Most of the vulnerabilities found by them can only be exploited under the same LAN without interaction with the cloud. Compared with the above fuzzers, SHADOWFUZZER is specific to MQTT clients, and it can discover vulnerabilities that can be triggered from anywhere on the Internet, which greatly increases the risk of these vulnerabilities.

9. Conclusion

We present the threat model of the trampoline-over-the-air attack and propose a black-box fuzzing system named SHADOWFUZZER to find vulnerabilities when IoT devices parse MQTT messages. SHADOWFUZZER leverages a shadow broker to transmit test cases sent from the fuzzer. It can detect the memory corruption and command injection vulnerabilities and avoid ethical issues. Attackers can leverage this type of vulnerabilities to perform trampoline-over-the-air attacks on MQTT clients that communicate with vulnerable brokers. By experimenting on 15 IoT devices, SHADOWFUZZER finds 34 zero-day vulnerabilities in 11 of them. All of the vulnerabilities can lead to remote control or denial-of-service attacks and affect multiple aspects of people's lives, such as transportation, home life, and office environment.

Acknowledgment

We thank the anonymous reviewers and Yingchen Fan, Yuxuan Wang, Yibing Qian, and Xianzi Kong for their helpful feedback of this work. This work is supported by National Key R&D Program of China (Grant No. 2021YFF03072), and National Natural Science Foundation of China (Grant No. U1936121). All opinions and findings mentioned in this paper are those of the authors.

References

[1] David Boswarthick, Omar Elloumi, and Olivier Hersent. *M2M communications: a systems approach*. John Wiley & Sons, 2012.

[2] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[3] Syaiful Andy, Budi Rahardjo, and Bagus Hanindhito. Attack scenarios and security analysis of mqtt communication protocol in iot system. In *Proceedings of the 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI), Yogyakarta, Indonesia, September 19-21, 2017*, 2017.

[4] Lucas Lundgren. Taking Over the World Through MQTT Aftermath. In *BlackHat 2017*, 2017.

[5] Martin Hron. Are smart homes vulnerable to hacking? https://blog.avast.com/mqtt-vulnerabilities-hacking-smart-homes, 2018.

[6] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, 2019.

[7] Wei Zhou, Chen Cao, Dongdong Huo, Kai Cheng, Lan Zhang, Le Guan, Tao Liu, Yan Jia, Yaowen Zheng, Yuqing Zhang, Limin Sun, Yazhe Wang, and Peng Liu. Reviewing iot security via logic bugs in iot platforms and systems. *IEEE Internet Things J.*, 8(14):11621–11639, 2021.

[8] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, 2019.

[9] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 2019.

[10] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. Cryptorex: Large-scale analysis of cryptographic misuse in iot devices. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, 2019.

[11] Yan Jia, Luyi Xing, Yuhang Mao, Dongfang Zhao, XiaoFeng Wang, Shangru Zhao, and Yuqing Zhang. Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, 2020.

[12] Trend Micro Research. The Fragility of Industrial IoTs Data Backbone. https://documents.trendmicro.com/assets/white_papers/wp-the-fragility-of-industrial-IoTs-data-backbone.pdf?v1, accessed: 2021-06-08.

[13] Araujo Rodriguez, Luis Gustavo, and Daniel Macêdo Batista. Program-aware fuzzing for mqtt applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[14] Prashant Anantharaman, Michael E. Locasto, Gabriela F. Ciocarlie, and Ulf Lindqvist. Building hardened internet-of-things clients with language-theoretic security. In *2017 IEEE Security and Privacy Workshops, SP Workshops 2017, San Jose, CA, USA, May 25, 2017*, 2017.

[15] F-Secure Corporation. mqtt_fuzz. https://github.com/F-Secure/mqtt_fuzz, accessed: 2021-06-08.

[16] Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, and Tahir Ahmad. MQTTSA: A tool for automatically assisting the secure deployments of MQTT brokers. In *Proceedings of the 2019 IEEE World Congress on Services, SERVICES 2019, Milan, Italy, July 8-13, 2019*, 2019.

[17] Santiago Hernández Ramos, M Teresa Villalba, and Raquel Lacuesta. Mqtt security: A novel fuzzing approach. *Wireless Communications and Mobile Computing*, 2018, 2018.

[18] Eclipse Foundation. Eclipse IoT-Testware. https://iottestware.readthedocs.io/en/development/smart_fuzzer.html, accessed: 2021-06-08.

[19] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 460–465. IEEE, 2020.

[20] Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. *Sensors*, 20(18):5194, 2020.

[21] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: toward a stateful network protocol fuzzer. In *International conference on information security*. Springer, 2006.

[22] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://boofuzz.readthedocs.io/en/stable/, accessed: 2021-06-08.

[23] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[24] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. *CoRR*, abs/2105.05445, 2021.

[25] Shadowfuzzer. https://github.com/ReAbout/ShadowFuzzer, accessed: 2022-01-29.

[26] Dick Hardt et al. The oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749, 2012.

[27] Alibaba Cloud. What is message queue for mqtt? https://www.alibabacloud.com/help/en/doc-detail/42419.html, accessed: 2022-01-24.

[28] Antonio Cilfone, Luca Davoli, Laura Belli, and Gianluigi Ferrari. Wireless mesh networking: An iot-oriented perspective survey on relevant technologies. *Future Internet*, 11(4):99, 2019.

[29] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, 2017.

[30] Dong Wang, Xiaosong Zhang, Ting Chen, and Jingwei Li. Discovering vulnerabilities in COTS iot devices through blackbox fuzzing web management interface. *Secur. Commun. Networks*, 2019:5076324:1–5076324:19, 2019.

[31] Roger A. Light. Mosquitto: server and client implementation of the MQTT protocol. *J. Open Source Softw.*, 2(13):265, 2017.

[32] mosquitto_subscribe. https://mosquitto.org/api/files/mosquitto-h.html#mosquitto_subscribe, accessed: 2022-01-24.

[33] CWE. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html, accessed: 2021-06-08.

[34] GTFOBins. https://gtfobins.github.io/, accessed: 2021-06-08.

[35] MQTT. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc385349257, accessed: 2021-06-08.

[36] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by internet-wide scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015.

[37] Censys. Search 2.0 Official Launch Announcement. https://search.censys.io/search/definitions?resource=hosts&sort=RELEVANCE&per_page=25&virtual_hosts=EXCLUDE&q=#MQTT, accessed: 2022-01-26.

[38] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. Looking from the mirror: Evaluating iot device security through mobile companion apps. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, 2019.

[39] Eclipse Paho. Eclipse Paho-MQTT and MQTT-SN software. http://www.eclipse.org/paho, accessed: 2021-06-08.

[40] PanguLab. Janus. https://www.appscan.io/, accessed: 2021-06-08.

[41] Androguard. https://github.com/androguard/androguard, accessed: 2021-06-08.

[42] Binary Ninja. https://binary.ninja/, accessed: 2021-06-08.

[43] OpenSSL - Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, accessed: 2021-06-08.

[44] die.net. iptables(8) - Linux man page. https://linux.die.net/man/8/iptables, accessed: 2021-06-08.

[45] Samsung. cotopaxi. https://github.com/Samsung/cotopaxi, accessed: 2021-06-08.

[46] CWE. CWE-704: Incorrect Type Conversion or Cast. https://cwe.mitre.org/data/definitions/704.html, accessed: 2021-06-08.

[47] GeekPwn. http://www.geekpwn.org/, accessed: 2021-06-08.

[48] Velop Whole Home Mesh WiFi: Strong WiFi Everywhere — Linksys. https://www.linksys.com/us/velop/, accessed: 2021-06-08.

[49] OWASP. Web Application Firewall. https://owasp.org/www-community/Web_Application_Firewall, accessed: 2021-06-08.

[50] Welcome to the OpenWrt Project. https://openwrt.org/, accessed: 2021-09-20.

[51] ZTE. Statement of Vulnerabilities in ZTE E8810/E8820/E8822 Series Routers. http://support.zte.com.cn/support/news/LoopholeInfoDetail.aspx?newsId=1014202, accessed: 2020-12-17.

[52] Lucas Lundgren. Light Weight Protocol Serious Equipment Critical Implications. In *Defcon 24*, 2016.

[53] Moshe Zioni. MQTT for Fun and Profit Explore Exploit. https://www.slideshare.net/moshez/mqtt-for-fun-and-profit-explore-exploit-owasp-il-2017-v12, 2020.

[54] M. S. Harsha, B. M. Bhavani, and K. R. Kundhavai. Analysis of vulnerabilities in MQTT security using shodan API and implementation of its countermeasures via authentication and acls. In *Proceedings of the 2018 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2018, Bangalore, India, September 19-22, 2018*, 2018.

[55] Syed Naeem Firdous, Zubair A. Baig, Craig Valli, and Ahmed Ibrahim. Modelling and evaluation of malicious attacks against the iot MQTT protocol. In *Proceedings of the 2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Exeter, United Kingdom, June 21-23, 2017*, 2017.

[56] Alessandra Rizzardi, Sabrina Sicari, Daniele Miorandi, and Alberto Coen-Porisini. AUPS: an open source authenticated publish/subscribe system for the internet of things. *Inf. Syst.*, 62:29–41, 2016.

[57] Ricardo Neisse, Gary Steri, and Gianmarco Baldini. Enforcement of security policy rules for the internet of things. In *Proceedings of the IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2014, Larnaca, Cyprus, October 8-10, 2014*, 2014.

[58] Aimaschana Niruntasukrat, Chavee Issariyapat, Panita Pongpaibool, Koonlachat Meesublak, Pramrudee Aiumsupucgul, and Anun Panya. Authorization mechanism for mqtt-based internet of things. In *Proceedings of the IEEE International Conference on Communication, ICC 2015, London, United Kingdom, June 8-12, 2015, Workshop Proceedings*, 2016.

[59] Meena Singh, MA Rajan, VL Shivraj, and P Balamuralidhar. Secure mqtt for internet of things (iot). In *Proceedings of the 2015 Fifth International Conference on Communication Systems and Network Technologies, Gwalior, India, April 4-6, 2015*, 2015.

[60] Ranbir Singh Bali, Fehmi Jaafar, and Pavol Zavarsky. Lightweight authentication for MQTT to improve the security of iot communication. In *Proceedings of the 3rd International Conference on Cryptography, Security and Privacy, ICCSP 2019, Kuala Lumpur, Malaysia, January 19-21, 2019*, 2019.

[61] Miao Yu, Jianwei Zhuge, Ming Cao, Zhiwei Shi, and Lin Jiang. A survey of security vulnerability analysis, discovery, detection, and mitigation on iot devices. *Future Internet*, 12(2):27, 2020.

[62] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, 2014.

[63] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[64] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013.

[65] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, 2018.

[66] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, 2020.

[67] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[68] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[69] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*, 2015.

[70] ARGAM ARTASHYAN. XIAOMI ENTERS HOTELS AND REAL ESTATE NICHES AND BRINGS AIOT SMART SOLUTIONS. https://www.gizchina.com/2019/11/19/xiaomi-enters-hotels-and-real-estate-niches-and-brings-aiot-smart-solutions/, 2019.

[71] This Is The New Brilliance V7 SUV For China. https://carnewschina.com/2017/10/19/new-brilliance-v7-suv-china/, accessed: 2021-06-08.

Appendix A.
Selected IoT Devices and Vulnerabilities Numbers

TABLE 4: 15 Selected IoT Devices.

Brand	Type	Model	Authentication	Broker Location
ZTE	Wireless Router	E8820	Hard-coded Certificate	Remote
TOPSEC	Access Point	TAP-62200	No	Remote
Alcatel-Lucent	Access Point	OAW-AP1101	No	Remote
HAN Networks	Access Point	AP211	No	Remote
Cetron	Access Controller	G500SE	Hard-coded Account	Remote
Rokid	Smart Speaker	Rokid Mini	Dynamic Account	Remote
DDPAI	Dash Recorder	X2S Pro	Hard-coded Account	Remote
Hikvision	NAS	H90	Hard-coded Account	Remote
Leapmotor	Vehicle	T03	Hard-coded Account	Remote
Microsound	Smart Speaker	Delta Mini	Hard-coded Account	Remote
Brilliance Auto	Vehicle Control	V7	No	Remote
Xiaomi	Smart Speaker	Xiao AI Speaker Pro	No	Embedded
Totolink	Wireless Router	T10	No	Embedded
SAIC-GM-Wuling	Vehicle T-Box	New Baojun	No	Embedded
Linksys	Wireless Router	Velop	No	Embedded

TABLE 5: CVE/CNVD/CNNVD Numbers of the Vulnerabilities. For the bug type, NPD means NULL pointer dereference, CI represents command injection, BO means buffer overflow, TE means incorrect type conversion.

Device	Device Type	No.	Vul Type
ZTE E8820	Wireless Router	CVE-2020-6881	NPD
TOPSEC TAP-62200	Access Point	CNVD-2020-73267	CI,BO,NPD[1]
Alcatel-Lucent OAW-AP1101	Access Point	CNVD-2020-73265	CI,BO,NPD
HAN Networks AP211	Access Point	CNVD-2020-73266	CI,BO,NPD
Cetron G500SE	Access Controller	CNVD-2020-73504	CI
Totolink T10	Wireless Router	CNVD-2020-28090	CI
Totolink T10	Wireless Router	CNVD-2020-28089	CI
Totolink T10	Wireless Router	CNVD-2021-43461	CI
Totolink T10	Wireless Router	CNVD-2021-43462	CI
Totolink T10	Wireless Router	CNVD-2021-43463	CI
Totolink T10	Wireless Router	CNVD-2021-44929	CI
Totolink T10	Wireless Router	CNVD-2021-44930	BO
Totolink T10	Wireless Router	CNVD-2021-44931	TE
Brilliance Auto V7	Vehicle	CNNVD-202012-189	BO
SAIC-GM-Wuling New Baojun	Vehicle	CNNVD-202012-149	CI
SAIC-GM-Wuling New Baojun	Vehicle	CNNVD-202012-149	BO

[1] CNVD assigned us only one number for multiple vulnerabilities. This situation also appears in the following two devices.

Appendix B.
GTFOBins Rules

TABLE 6: GTFOBins rules we collect.

Index	Payload	Command
1	`. -o ! -name . -exec COMMAND`	find
2	`--exec='! COMMAND '`	mail
3	`-x sh -c 'reset; exec COMMAND 1>&0 2>&0'`	watch
4	`-e 'os.execute(" COMMAND ")'`	lua
5	`-s --eval=$'x:\n\t-'" COMMAND "`	make
6	`COMMAND`	nohup,busybox,env
7	`-r 'system( COMMAND );'`	php
8	`-u / COMMAND`	flock
9	`'BEGIN system(" COMMAND ")'`	awk,gawk,mawk
10	`-e '/bin/sh -c COMMAND ' rdoc`	gem open
11	`-e "exec('/bin/sh -c COMMAND ')`	jrunscript
12	`--dev null --script-security 2 --up '/bin/sh -c COMMAND '`	openvpn
13	`-e 'exec "/bin/sh -c COMMAND ";'`	perl
14	`-c 'import os; os.system("/bin/sh -c COMMAND ")'`	python
15	`-p ''/bin/sh -c COMMAND 1>&0''`	rake
16	`-e '/bin/sh -c COMMAND ' 127.0.0.1:/dev/null`	rsync
17	`-e 'exec "/bin/sh -c COMMAND "'`	ruby
18	`-cf /dev/null /dev/null --checkpoint=1 --checkpoint-action=exec=' COMMAND '`	tar
19	`-a /dev/null COMMAND`	xargs
20	`-n 'le exec COMMAND ' /etc/hosts`	sed

Appendix C.
Case Study

Xiaomi Xiao AI Speaker Pro. A `Mosquitto` broker without authentication was embedded in Xiaomi Xiao AI Speaker Pro for inter-process communication, and the process `mibt_mesh_proxy` receives messages published from `mibt_mesh`. A command injection vulnerability occurs when the subscriber processes messages. Attackers located in the same LAN can publish a crafted payload to this intelligent speaker to gain the root privilege of the device. This speaker is top-rated and has been used in more than 5,000 hotels, covering at least 34 cities [70]. As Figure 6 shows, attackers staying in these hotels can control the speakers in other rooms via this vulnerability if there is no AP isolation deployed.

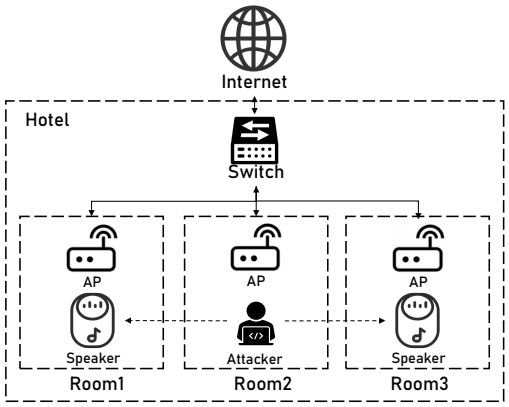

Figure 6: A potential attack scenario of Xiaomi AI Speaker Pro. The attacker living in the hotel can attack the speakers in other rooms if there is no AP isolation.

Cetron Access Point and Access Controller. The Cetron Beecon series access points and some access controllers such as G500SE and G1000S can be controlled by a WeChat mini-program, which communicates with the devices over the plaintext MQTT protocol. The credential is shared among all these devices and is hard-coded in the firmware. Three command injection vulnerabilities we found in these devices are harmful. An attacker can append a baleful command to a string variable, such as `telnetd` to open the telnet service.

Totolink T10 Wireless Router. An MQTT broker based on `mosquitto` is embedded in Totolink's multiple wireless routers such as T10. The broker is in charge of distributing HTTP data sent from the Web interface to the handlers. The process `cste_sub` subscribes to the topics of the HTTP messages and handles them. Our tool discovered six command injections, one incorrect type conversion, and one buffer overflow in the parsing of MQTT payloads.

Brilliance Auto V7. The Brilliance V7 [71] is a mid-size CUV (crossover Sport Utility Vehicle) produced by Brilliance Auto under the Zhonghua brand. The vehicle's IVI (In-Vehicle Infotainment) system receives location messages published from a broker that can be accessed anonymously. A stack overflow vulnerability exists in the process of string replacement, leading to a DoS attack or a remote command execution attack.

SAIC-GM-Wuling New Baojun. New Baojun is a Chinese automobile marque owned by a joint venture of General Motors and SAIC-GM-Wuling Automobile. There is an MQTT broker embedded in this vehicle's T-Box (Telematics Box) system. Attackers in the in-vehicle network can access the broker with no credential. An attacker can trigger a command injection vulnerability and gain a root shell by publishing an exploit message to the broker after accessing the in-vehicle network.