IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing

Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiaofeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, Kehuan Zhang

Presented by Sezana Fahmida
Outline

• Introduction
• Background
• Challenges
• Scope & Assumptions
• Design
• Implementation & Evaluation
• Discussion
• Conclusion
Introduction

• Internet of Things (IoT) dominating the global market
• IoT devices is projected to reach 20.4 billion in 2020, forming a global market valued $3 trillion
• smart plugs, smart door locks, smart bulbs etc
• 2014 to 2016, 90+ independent IoT attack incidents
• Targets implementation flaws within a device’s firmware
Background
Typical IoT architecture

Fig. 1: Typical smart home communication architecture
Typical IoT architecture

- Devices equipped with sensors
- Wireless Connection
- IoT app to control devices provided by vendors
- Communication mode between app and device can be
  - Direct (wifi/Bluetooth)
  - Delegated (via a cloud server)
Obstacles in Firmware Analysis

• Firmware: Special software providing
  • System control
  • Status monitoring
  • Data collection

• Highly customized to fit device architecture

• Main Challenges
  • Firmware Acquisition
  • Firmware Unpacking
  • Executable Analysis
Motivation

• Skip direct firmware analysis by alternative approach
• Intuition: Leverage IoT apps to find vulnerabilities
• Advantages:
  • No need for firmware analysis
  • Avoids reverse engineering binary executables
  • Feasable: Most IoT devices use app
• Design goal: generate protocol-guided and cryptographic consistent fuzzing messages from IoT apps to find memory corruption
Challenges in IoT Fuzzer Design

• Mutating fields in networking messages
  • Device specific protocols are used

• Handling encrypted messages
  • Communication between app and device encrypted
  • Code obfuscation
  • Increases complexities

• Monitoring crashes
  • Cannot locally monitor the running process in the system
Solutions

• Mutating fields in networking messages
  • Mutate data at the source

• Handling encrypted messages
  • Reusing cryptographic functions at runtime

• Monitoring crashes
  • Use heartbeat mechanism
Scope & Assumption

• IoT devices with apps
• Communication channel: Wifi
• Direct Connection, No cloud server
• Android platform
IoTFuzzer Design

- Two phases
- App analysis
  - UI analysis
  - Data Flow analysis
- Fuzzing
  - Runtime mutation
  - Response Monitoring
App Analysis

Picture taken from author’s slides
App analysis

• UI analysis
  • Static analysis of apk
  • determine the UI elements that eventually lead to the message delivery
  • from the target network communication APIs construct the backward code paths to UI event handlers
  • Activity transition graphs: To find the order of events
App analysis

• Data flow analysis
  • to recognize the protocol fields and record the functions that take these arguments
  • Dynamic taint tracking
  • Taint source: string, system API, user input
  • Taint sink: networking API and encryption functions
Fuzzing

Fig. 2: Overview of IOTFUZZER
Fuzzing

• Runtime Mutation
  • Dynamic Function Hooking
  • Intercept function calls and mutate the function arguments
  • Fuzzing Scheduling
  • Only mutate a subset of function parameters
  • Fuzzing policy
    • Changing the lengths of strings
    • Changing the integer, double or float values
    • Changing the types or provide empty values
Fuzzing

- Response monitoring
- Device status inferred from IoT device responses
  - Expected Response
  - Unexpected Response – Error is triggered
  - No Response - Error may be triggered
  - Disconnected – System crash
Fuzzing

• TCP-based connection: look for disconnection
• UDP-based connection: send heart-beat message from app
Implementation

- 17 representative IoT devices from different categories
Evaluation

• 15 serious vulnerabilities (memory corruptions) in 9 devices.

<table>
<thead>
<tr>
<th>Device</th>
<th>Vulnerability Type</th>
<th># of Issues</th>
<th>Remotely Exploitable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belkin WeMo (Switch)</td>
<td>Null Pointer Dereference</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>TP-Link HS110 (Plug)</td>
<td>Null Pointer Dereference</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>D-Link DSP-W215 (Plug)</td>
<td>Buffer Overflow (Stack-based)</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>WD My Cloud (NAS)</td>
<td>Buffer Overflow (Stack-based)</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>QNAP TS-212P (NAS)</td>
<td>Buffer Overflow (Heap-based)</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Brother HL-L5100DN (Printer)</td>
<td>Unknown Crash</td>
<td>1</td>
<td>Not determined</td>
</tr>
<tr>
<td>Philips Hue Bridge (Hub)</td>
<td>Unknown Crash</td>
<td>1</td>
<td>Not determined</td>
</tr>
<tr>
<td>WD My Passport Pro (NAS)</td>
<td>Unknown Crash</td>
<td>1</td>
<td>Not determined</td>
</tr>
<tr>
<td>POVOS PW 103 (Humidifier)</td>
<td>Unknown Crash</td>
<td>1</td>
<td>Not determined</td>
</tr>
</tbody>
</table>
Fig. 5: Fuzzing Accuracy
Discussion

• Provides high specification coverage, low code coverage
• Does not consider cloud relay
• cannot generate memory corruption types and root causes directly
• final vulnerability confirmation always requires some kinds of manual efforts.
• False positives & negatives
Conclusion

- IoTFuzzer - first IoT fuzzing framework
- Protocol guided fuzzing achieved without protocol specifications
THANK YOU!!!