Security Analysis of Wearable Smart Health Devices and Their Companion Apps

Daniel Timko, Mike Sharko, and Yanyan Li
Department of Computer Science and Information Systems, California State University San Marcos, USA
timko002@csusm.edu, shark003@csusm.edu, yali@csusm.edu

***Abstract*—Wearable Smart Health Device (WSHD) went from being a fantasy to reality in the past few years with the drastic development in the new smart technology. Just like any other technologically developed devices, along with its positive impact on the society, WSHD comes with its own risks as well. In this paper, we focus on analyzing how WSHD can leak the personal information of end users through identification of vulnerabilities in the devices, their companion apps and their communication channels. Through the static and dynamic analysis of the companion apps, we discovered exposed security keys, authentication tokens and API endpoints that can be used to modify the data sent from the app to the cloud. We were also able to spoof Bluetooth Low Energy data packets to change the attributes on the WSHD. Our study sheds light on how severe these security risks are to the smart health device manufacturer as well as the user. To help mitigate these risks, we have proposed a few recommendations that the manufacturers and developers could follow to address the security threats in their products.**

I. Introduction

A Wearable Smart Health Device (WSHD) refers to physical devices embedded with sensors or actuators that monitor the wearers health and are connected to the internet to exchange data. These wearable devices have spread at a fast pace through many sectors, enabling users to track their fitness and monitor their health. WSHDs are not only changing how users connect to their surroundings, but it is also transforming industries as a tool to reduce costs and increase efficiency. The Wearable Healthcare Device market has grown substantially over the past decade, with its size expected to more than double in the next 10 years according to Straits Research [1]. This exponential growth and demand of WSHD often leads to rapid design and deployment of products, and many times, security is a treated as secondary concern. Due to this rapid development, WSHD are frequently deployed with vulnerabilities that make them susceptible to different types of attacks and data leakage. As a result of the interconnected nature of smart devices into systems and infrastructures, these systems can be used as the weak entry point for attacks [2], [3].

Typically, a smart device utilizes a mobile companion application to access the internet. The mobile application pairs and bonds with the WSHD to establish a communication channel to send and receive data. This allows the user to control the device, adjust its settings, and provides an interface with the device. The communication protocols employed by smart devices, such as Bluetooth Low Energy (BLE), generally utilize low energy consumption to overcome their battery constraints [4]. However, the limited computing and battery resources available to these devices hinder the security of their wireless communications. This is particularly significant considering the mobile companion app's role as an important component of the WSHD system, as it has access to a user's sensitive data [5], [6].

In this work, we focus on the potential for exploitation of WSHDs. We selected a number of WSHD from different brands and device types to examine their corresponding companion apps. In our app analysis, we reverse engineer the applications and then perform both a static and dynamic analysis of the devices. Afterwards, we assess the risks of personal information exposure resulting from the lack of adherence to best security practices. Furthermore, we leverage the information gained from our analysis to test its vulnerability in an attack scenario.

This paper makes the following contributions:

1) We reverse-engineer the companion apps of a variety of wearable smart health devices to investigate their communication protocols and security implementations.
2) The analysis of the communication between the WSHD, their companion app, and the cloud reveal insecure data handling and personal information leakage.
3) Our analysis of companion apps and WSHD devices reveals keys, API endpoint information, and encryption schemes that can be used to access backend servers.

II. Related Works

A. WSHD Companion Apps

WSHD are a type of IoT device that is worn on the body and helps to monitor health attributes of the wearer. These WSHD often have a companion application that functions as a mediator for the user-device-service usage of the gadget-application-cloud communication architecture. Companion apps serve as the user interface, providing access to the WSHD's features by communicate wirelessly with the device. Although security is critical, it is also challenging to implement with the rapid growth of IoT devices, leaving a wide surface for attacks over wireless communication [7]. The constraints of computational, battery, and memory resources on IoT devices limit the ability to encrypt data before transmission between the device, application or with the cloud [8]. Consequently, an analysis of the mobile companion apps can provide valuable insights into the security of the underlying IoT devices, as they often contain code that is used to communicate with the devices and control

Levels	Attack Vectors
Passive Attacker	(4) Device to App Communication, (5) App to Cloud Communication
Active Attacker	(1) WSHD, (2) Companion App, (3) Cloud

TABLE I
LEVELS OF ATTACKERS AND THEIR ATTACK VECTORS

their functionality [9]. These mobile companion applications also serve as the gateway between the WSHD and the cloud, for the purpose of data collection. This architecture has several advantages, such as ease of use and reduced cost. However, it also introduces security challenges in authenticating and encrypting communication [10]. Additionally, companion applications can leak personal data through insecure communication with the wearable device [11], [12]. Lyons et al. explored the security risks of logging sensitive data in applications, and discussed how apps leak that information [13].

B. Security Challenges

There are many security challenges in the WSHD environment, on the information level (data security, integrity, confidentiality, and privacy), access level (control, authentication, and authorization), and functional level (network resilience, and IoT self organization). Attacks can occur on different layers using IoT devices as an entrance gate to the network, or to limit, misuse, and extend IoT functionalities [14]. Jr et al. conducted an analysis on IoT companion apps of the best 96 selling IoT devices on Amazon found that many devices share the same mobile application, even devices from different manufacturers, and 50% of the applications that serve 38% of the devices had security flaws on the app-device communication [15]. Neshenko et al. presented an extensive survey of previous researches identifying IoT vulnerabilities at different layers: device, network, and software. The main vulnerabilities identified are inadequate physical security and power harvesting, lack of or weak authentication and/or encryption, excess of open ports, inappropriate patching, inadequate access control and/or audit procedures [16].

III. THREAT MODEL

In this study, we aim to review the exploitable aspects of WSHD and their companion applications. Our threat model in Fig 1 shows all the components in our system: WSHD, companion apps, cloud, and communication protocols. In this research, we investigate the security risks involved in the communication established between the WSHD (1), its companion application (2), and the cloud (3). We also focus on device-to-app communication (4) and app-to-cloud communication (5). Furthermore, we consider an attacker who can gain access to the mobile device hosting the companion app, potentially leaking personal data. The attack vectors are defined in Table I.

A passive attacker can silently listen and capture data packets from the target network communicating between the WSHD and the companion app (4) on the mobile device using BLE. Attackers can use sniffing to steal sensitive user information or commit identity theft. When the companion apps collect or update data from communication with the device, that data is often transferred to the cloud (3). We also investigate the security risks in app-to-cloud communication (5), typically conducted over Wi-Fi or mobile data.

In addition to the threats of a passive attacker, we are also studying the vulnerabilities available to an active attacker, potentially able to capture and manipulate traffic with the WSHD (1) and cloud (3). We apply static analysis to identify vulnerabilities in the coding style and practices followed by the developers on the companion app (2). From this information, attacks can be conducted through communication protocols to inject their own malicious commands to the WSHD (1) or altering data packets to the cloud (3).

We use dynamic analysis to capture and obtain valuable information from the network traffic and system logs of the companion app, cloud, and the WSHD during active use. Using static analysis, we found API endpoints and API keys exposed in the code through reverse engineering. This information could potentially be exploited by an attacker to create valid API calls that can potentially steal or manipulate user data through unauthorized access. Additionally, we were able to identify encryption in the BLE communication between the device and app. When apps use weak or no encryption schemes, an attacker can potentially compromise the encryption and decipher your device's traffic.

Access to this crucial data required no special privileges, making it equally available to potential attackers aiming to exploit the system for their own purposes. Thus, the user falls prey to these attacks with no means to knowing what and how it all happened.

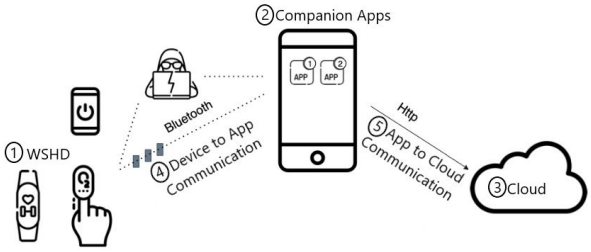

Fig. 1. Threat model showing all the components

IV. METHODOLOGY

In this research we have explored several aspects of security in companion apps. These aspects can be broken down into 3 sections, 1) Secure communication between the device and the companion app through its BLE commands. 2) Secure Communication between the companion app and web services through API endpoints. Lastly, 3) Security between the companion app and other apps on the device.

A. Design Overview

We utilized several approaches to analyze the security risks of the communication in our device's companion apps. In this work, we focus on Android applications. First, we collected the apk files of the companion apps associated with the devices we used for this research. Next, we reverse engineered the applications using .apk decompilers. This gave us valuable information on the API endpoint calls of the companion apps,

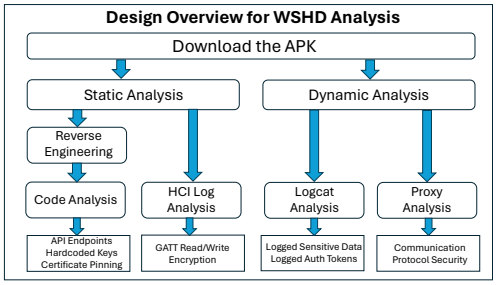

Fig. 2. Overview of the research procedure.

the structure and use of its BLE communication, and the cryptography use including hashing algorithms, keys and token generators. Afterwards, we performed static analysis through observing logs of communication between the devices and our companion apps, as well as a review of the decompiled code. Lastly, we performed a dynamic analysis of the companion apps through Android's logcat and by setting up a proxy and intercepting communication between the web server and the companion app. Logcat reveals what kind of information the application is leaking through regular use, and by intercepting communication through a proxy will reveal the communication vulnerability between the app and the server hosting its API endpoints. A full overview of our design can be found in Fig 2.

B. Ethical Consideration

All testing was conducted on devices owned by the research team, using newly created test accounts dedicated to this study. Study data, including screenshots, were collected via these test accounts to prevent private data leakage. The results were shared with the manufacturer of each tested device and companion app.

WSHD Table				
Category	Type	Brand	Model	Companion App
Fitness	Tracker	Fitbit	Alta	Fitbit
Fitness	Tracker	Garmin	Forerunner 45	Garmin Connect
Health	Finger Oximeter	LPOW	Fingertip Pulse Oximeter Sp02	LPOW Pulse Oximeter
Health	Heart Rate Monitor	Polar	H10	Polar Beat
Health	Blood Sugar	Accu-Chek	Guide	mySugr

TABLE II
DEVICES ANALYZED IN THIS STUDY.

C. App collection

Before diving into the implementation, comes the crucial step of choosing which WSHD to use for this study and their mobile companion applications to analyze for vulnerabilities. Since the first phase of this study was done on a small scale, a manual approach was taken to choose the WSHD and associated companion apps. We searched for popular WSHD with companion apps from different categories including heart rate monitors and fitness trackers [17], [18]. Upon gaining insight into the various categories and the companion apps that constitute each, we went on to randomly select 5 devices to use as a part of this research.

The companion applications chosen for analysis were the ones referenced on each WSHD's Amazon.com store page for the device. Four devices from different categories were used to cover diverse fields. Garmin Connect (V4.53) was chosen to identify vulnerabilities in WSHD fitness tracking devices. To further enhance the process of identifying vulnerabilities, two additional companion apps were chosen to establish similarities in vulnerabilities among health monitoring devices. Polar Beat (V3.5.7) - a heart rate monitoring companion app, and Mysugr (V3.95.0) – a blood sugar monitoring companion app were used to discover similar vulnerabilities among health monitoring apps. The Finger Oximeter-SpO2 (V1.3.2) - an an oxygen monitoring app, serves as the example to identify vulnerabilities in WSHD general health. We further explored the similarities in vulnerabilities among health monitoring apps and fitness tracking apps by using Fitbit (V3.58) and the above mentioned health monitoring apps. The full list of chosen companion apps and WSHD devices can be found in Table II.

D. Implementation

To analyze the Android applications, we obtained the .apk files from apkpure [19] and Uptodown [20]. These files were decompiled into .dex files using the reverse engineering tools Apktool [21] and jadx [22]. Apktool decompiled the .apk files into smali, providing insights into the application's architecture and hierarchical structure, while jadx decompiled the entire application into .java and .kt files, offering visibility into modules and flow control. Additionally, jadx facilitates the conversion of DEX/Smali code to JAVA source code, making it a valuable tool for static analysis. We then utilized jadx's GUI to explore the source code of the .apk files, enabling examination, editing, and vulnerability identification.

The decompilation process involves placing the .apk android file for each companion application in the same folder as apktool. Using the 'apktool d <filename>' command, where filename is the name of the android file, we initiate the decompilation process. After decompilation, a folder containing the application files (Original, res, smali), as well as AndrodManifest.xml and apktool.yml files, is generated. In this folder named after the android application, we examine the decompiled files for security vulnerabilities such as API keys or API calls, and insecure passwords.

After reverse engineering, we looked for passwords, encryption or API keys in the decompiled code. We browsed the code in Android Studio to search the project files for hard coding issues where developers may store API Keys, passwords and API links in plain text without encryption. We used keyword matching to search within the text for "_API, _KEY, secret, pass, cipher, http, hts/, tp:, _Token, UUID". We found that several api links were obfuscated by splitting the link into two strings, hence "hts/" and "tp:", which results from splitting "https:/". When we found the API link, we also included the link keywords in the search and then scanned the pages the API links were used in. Using jadx we took apk files of our companion apps to reverse engineer the .dex files, and we be able to see the Java source code that these .apk files are running on. We also used android debug bridge (ADB) to monitor the logcat output, analyze configuration files and browse the app's database in order to find unencrypted passwords, API calls with unencrypted tokens.

In addition to using static and dynamic analysis, we also conducted BLE sniffing on the device communication traffic. For this, we used the following programs: "Homepwn", and "nRFConnect" to audit the BLE and WIFI of the devices. To begin, we detect devices located nearby. After their discovery, connect to them and see their BLE characteristics. Knowing the UUID of an attribute, we can attempt to write a specific value to attributes with WRITE property. This will change its current value to a new one.

E. App analysis

We utilized both static and dynamic analysis tools to detect vulnerabilities in our companion apps. In this work we focused on the security of 5 devices and their corresponding companion apps.

1) Static analysis: In the static analysis phase, we studied the companion apps as a standalone entity. we were able to retrieve some valuable information like the API keys used by the API endpoints, and the encryption techniques used by the developers, etc. To locate this information, our primary strategy was manual code review, combined with the use of keyword patterns to search through the decompiled code. We also utilized BLECryptracer [23], [24] to assist in locating BLE get and set commands along with their encryption. The encryption technique further helped us to decrypt certain encrypted information that were hardcoded in the application files. By analyzing three different companion apps from the same category of health monitoring devices, we were able to discover and test out some common vulnerabilities among them. We were also able to identify similar vulnerabilities among health monitoring and fitness tracking apps as they all fall under the same umbrella of WSHD.

2) Dynamic analysis: We used two different approaches for our dynamic analysis. The first approach was to analyse our companion apps in logcat. Through logcat we were able to easily identify internal information to the application and the api endpoints including authorization tokens, JSON callback data and post encryption user ID variables. With this data we were able to create our own calls to the API endpoints in several applications.

Next, we set up a proxy to intercept and analyze additional communication with our companion app. We used burp [25] to set up a proxy to funnel our app's traffic through. Additionally, we installed the burp user certificate, which breaks TLS encryption, allowing us to view unencrypted traffic. For the purposes of this work we do not install the certificate into the systems directory. This is done specifically to test for vulnerabilities in the certificate pinning techniques within our companion app.

V. Results

After reverse engineering companion applications, we discovered the following vulnerabilities: Insecure API keys were found almost in every companion app. API calls and API tokens were stored in plain text without encryption. User sensitive data was kept in plain text without encryption. Using applications for BLE sniffing and MAC spoofing, we were able to detect GATT attribute vulnerabilities, which is critical information when it comes to sensitive data.

A. Evaluation

Our research targets the security aspect of the WSHD system communication with focus on information and access. On the information level, we evaluated if the WSHD systems preserve data integrity with no alterations, secure the source identity to guarantee anonymity, block access of third-parties to preserve confidentiality, and check if the user information is not disclosed to ensure privacy. The evaluation target on the access control level is to ensure that only legitimate users have access, the authentication to check if the device has access to the network, and the authorization to confirm if only authorized users have access to devices.

We analysed traffic from the device to its mobile companion app in five different devices, and performed static and dynamic analysis of the mobile application. Although we were able to monitor the communication traffic, some applications had implemented data encryption and we did not have access to the data being transmitted. We found that most of the companion apps had some vulnerabilities, such as exposed keys, and weak protection of API requests, with hard coded API endpoints and their respective access tokens leaking. The full results of our analysis can be found in Table III.

B. Findings

Through the use of HCI snoop logs we were able to view the activity of the Bluetooth low energy (BLE) devices. These logs were then downloaded from android devices and analyzed using Wireshark. From our preliminary analysis, we encountered several levels of encryption in each device. Once connected and bonded to the device, we were able to view general attribute profiles (GATT) and with the nrfConnect application, and attempt to send new attributes to the devices. To the best of our knowledge, we were able to identify the encryption of the companion apps.

In our analysis of the communication between devices and the cloud, we found that each companion app implemented certificate pinning; however, Webview related traffic within the apps accepted any certificate. Later, we revisited the decompiled code to confirm the implementation of certificate pinning by searching for common implementations, such as an extended x509 classes with the checkServerTrusted method.

Additionally, we were able to identify the cryptography functions and libraries used for several more of these web service calls, even if we were not able to obtain the tokens.

Fitbit Alta. Once connected and bonded, we found that the Fitbit Alta didn't encrypt some read or write BLE packets, allowing us to change device values such as the owner's name using plain text BLE commands. We also found that we were able to connect and bond with the device without having to physically accept the connection on the device itself. When we analyzed the decompiled application, we found several exposed API keys as well as their google storage bucket link.

Component	Analysis	Fitbit Alta	Garmin Forerunner 45	LPOW Pulse Oximeter Sp02	Polar H10	Accu-Chek Guide
2) Companion App	Visible API Endpoints	Yes	Yes	Yes	Yes	Yes
	Hardcoded API Keys	Yes	Yes	Yes	Yes	No
	Cert. Pinning Unimplemented	No	No	No	No	No
4) Device to App Communication	Unencrypted GATT cmd. (Read)	Yes	No	No	No	No
	Unencrypted GATT cmd. (Write)	Yes	No	Yes	No	No
5) App to Cloud Communicaiton	Logged Sensitive Data	No	No	Yes	No	Yes
	Logged Auth Tokens	No	No	Yes	No	Yes

TABLE III
THE RESULTS OF THE STATIC AND DYNAMIC ANALYSIS

We did not find any leaked personal information or leaked authorization codes in the logcat data.

Garmin Forerunner 45. Our analysis of the BLE communication found that some read and write GATT attributes were encrypted into Byte strings. The device itself also required the wearer to confirm bonding during our connection attempt, allowing users to be notified of attempts to snoop on the traffic or send BLE attributes. Additionally, we were able to connect to the device directly using nrfconnect and directly change attributes, such as the device name and update GATT Heartbeat attribute. An example of GATT attributes being updated can be seen in Fig 7. We did not find any leaked personal information or leaked authorization codes in the logcat data.

LPOW Pulse Oximiter Sp02. The oximeter, a WSHD that transmits very sensitive data of the user, sends the heart rate BLE sensor readings in plain text. The API endpoints for this application can be viewed in Fig 3. Among the Apps tested, we also found that it was typical for them to have a separate encryption scheme for the API endpoints and the BLE communication. In Fig 4 we can see the encryption algorithms mentioned in the finger oximeter SP02 app. In this case, we found that RSA encryption was used with connection to the API endpoints. We also identified the existence of insecure and weak algorithms, such as MD5 hash, which has been used to cause collision in order to forge validated digital certificates.

In the companion apps, we noticed that logcat was leaking the authorization tokens and callback data returned from their API endpoint during normal activity of the application. This leaked callback data leaked personally identifying information about users, an example of which can be found in Fig 5.

Polar H10. Polar Beat encrypts its BLE traffic with an encryption key. For its web communication they use a ciper suite with an array of defined encryption schemes available for TLS communication. Upon further exploration, it appears the communication with the API server utilizes AES-128 bit GCM encryption with a SHA256 hash. We were able to find exposed API keys, including their google API Key and access information for their Firebase database.

While authorization tokens were not present in the apps logs, they did log the ID of the user connected to the devices. This ID, in conjunction with the users' email and password, could be used to access the stored physical information of users by connecting to their API endpoint.

Accu-Chek Guide. We found that the mySugr app uses a block cipher called LOKI as well as 128 bit AES encryption in CCM mode for their BLE encryption schemes between the device and the APP. While we were able to find the API endpoints, we were not able to find any exposed API keys.

The app uses a marketing tracker called Braze. We found that information regarding the API key, user_id and api endpoint were being constantly broadcast in logcat. Additionally, the structure of the API request was also logged in the logs, potentially allowing people to make unauthorized requests.

Fig. 3. API endpoint calls found in the finger oximeter SpO2 app.

Fig. 4. Encryption schemes within finger oximeter SpO2.

VI. DISCUSSION

App security is crucial to businesses and individuals that are working with customers' data and third-party services. WSHD Security is important because it protects users and their sensitive data regarding their health, bank accounts, or personal correspondence. In the following section, we are going to discuss the implications of our security WSHD and companion app findings to provide recommendations on how to improve WSHD security. Based on our research, we hope to provide information on security that can be adopted and used in future projects.

A. Implications & Recommendations

In this work, we were able to find API endpoints, hardcoded keys and encryption methods through reverse engineering the .apk files. We understand that there is no way that would completely protect an application; however, there are steps that can make it more difficult to attack.

Code obfuscation and API key protection are vital measures to enhance application security. Decompiling applications from sources like APKMirror [26] exposes sensitive information, including API keys, which attackers can exploit to access endpoints and incur significant costs for app owners. Implementing code obfuscation, using the R8 tool to compress java code into optimized dex code, can both optimize the code, and also make it more difficult for humans to interpret.

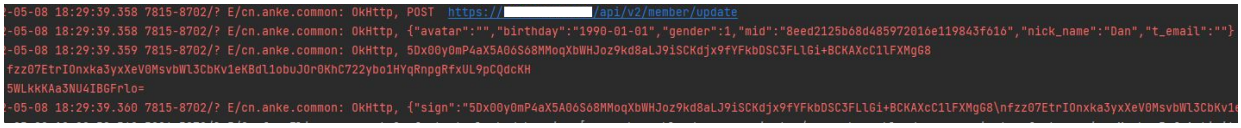

Fig. 5. Example of a callback leaking PII and authentication signature. Information displayed here is from our test user account.

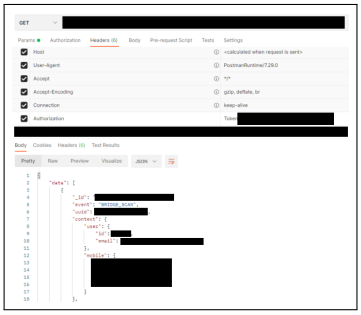

Fig. 6. Example accessing backend server with authentication information.

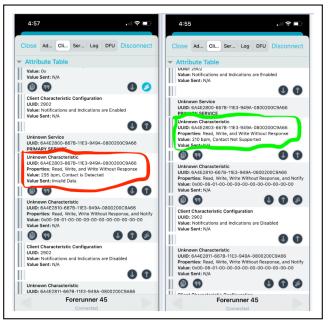

Fig. 7. Garmin watch GATT value change.

API keys, commonly used in compiled code for rapid application development, pose another major security risk. In our results, we found hardcoded API keys for several of the tested applications. The consequence of exposing these keys is that they risk the potential for unauthorized access to services or resources. Encrypting these API keys is a simple and common way to render them more resistant to exploitation. Alternatively, API keys can be pulled from an external server when the services in the application need them.

Additionally, sensitive data such as access tokens or personal data should not be logged through logcat. Combined with information about the API endpoints, unauthorized access to the backend server can be made to potentially retrieve or alter data. In our results, we found examples of authentication codes and endpoint information in the tested apps. Using our findings, we were able to replay authentication codes leaked in the logs to access personal account and device information through a postman request to the backend server of a WSHD, shown in Fig 6. Details about the device and app used were redacted upon request.

Its important to note that our findings, while focused on the security risks of a select number of WSHD devices and their companion apps, have broader implications. Through methods like malware distribution, or other malicious applications on the mobile device, an attacker could exploit similiar vulnerabilities to gain unauthorized access to harvest user data. Additionally, while logcat access is only available to privileged system apps, it is still prone to data leakage [27], enabling the access to sensitive information found in this work.

B. Limitations & Future Work

One of our primary limitations is the number of devices tested. We recognize that to get a full understanding of how often these vulnerabilities are present in modern companion devices, we would need to analyze more devices. In a future work we could automate the static and dynamic analysis to cover more devices.

Based on our analysis of the companion apps, we identify that a MAC spoofing attack would be particularly potent against devices similar to the Finger Pulse Oximeter SP02. The BLE communication to the device is not encrypted and it should be possible to spoof the device through the creation of a GATT server that is sending manufactured notifications of heart-rate or blood pressure readings to the companion app. Through this, an attacker would be able to submit any readings they wanted to the device, or to repeatedly send commands to the device in order to perform a DOS attack.

Our proxy analysis of the companion apps revealed the use of certificate pinning to block the TLS handshake of spoofed certificates. We surmise that this could be overcome through installing the certificate into the system's trusted certificate folder. A better understanding of the vulnerabilities of the API endpoint communication could be gained from analyzing them under this condition. We recognize that in order for an attacker to access these log files in order to gain API authorization would require 'signatureOrSystem' level permissions. Future work could analyze malware that might compromise apps on the system with these permission to leverage a co-located attack on the app using the read_logs permission to access the logcat files. Lastly, based on our static analysis of the code we found several security vulnerabilities such as admin mode checks. Future analysis could be done on the threat surfaces available to an attacker that was able to make small code changes and then recompile the companion app.

VII. Conclusion

We conducted a thorough analysis of five companion apps and uncovered exposed keys, API endpoints, and other vulnerabilities. These findings shed light on the security implementations and encryption systems employed by these apps. By closely examining the communication channels, we gained valuable insights into the potential security risks and vulnerabilities present. Our analysis revealed that some API endpoints lacked sufficient protection, making them susceptible to various attacks. We discovered multiple encryption types used in communication, with separate implementations of the same encryption scheme across different companion apps.

We also uncovered potential risks associated with the use of sniffing applications and log leakage. Through sniffing, we were able to gain enough information to modify unencrypted attributes and introduce new information on the devices. Through log leakage we were able to see API callbacks and access authentication signatures. Based on our comprehensive analysis, we strongly recommend improvements in the security measures implemented by both the companion apps and the WSHD themselves.

References

[1] "Wearable healthcare devices market size, demand, report to 2032," https://straitsresearch.com/report/wearable-healthcare-devices-market.

[2] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[3] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster, "Web-based attacks to discover and control local iot devices," in *Proceedings of the 2018 Workshop on IoT Security and Privacy*, 2018.

[4] A. M. Robles-Cordero, W. J. Zayas, and Y. K. Peker, "Extracting the security features implemented in a bluetooth le connection," in *IEEE International Conference on Big Data (Big Data)*, 2018.

[5] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[6] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[7] J. Classen, D. Wegemer, P. Patras, T. Spink, and M. Hollick, "Anatomy of a vulnerable fitness tracking system: Dissecting the fitbit cloud, app, and firmware," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 2018.

[8] A. Barua, M. A. Al Alamin, M. S. Hossain, and E. Hossain, "Security and privacy threats for bluetooth low energy in iot and wearable devices: A comprehensive survey," *in IEEE Open Journal of the Communications Society*, 2022.

[9] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Looking from the mirror: Evaluating {IoT} device security through mobile companion apps," in *28th USENIX Security Symposium*, 2019.

[10] H. Liu, J. Li, and D. Gu, "Understanding the security of app-in-the-middle iot," *in Elsevier Computers & Security*, 2020.

[11] D. Yeke, M. Ibrahim, G. Tuncay, H. Farrukh, A. Imran, A. Bianchi, and Z. Celik, "Wear's my data? understanding the cross-device runtime permission model in wearables," in *2024 IEEE Symposium on Security and Privacy (S&P)*, 2024.

[12] M. Tileria, J. Blasco, and G. Suarez-Tangil, "WearFlow: Expanding information flow analysis to companion apps in wear OS," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[23] "Tool to identify the presence of cryptographically-processed ble characteristics in android apks." https://github.com/projectbtle/BLECryptracer.

[13] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. Tapiador, S. Egelman, N. Vallina-Rodriguez *et al.*, "Log: It's big, it's heavy, it's filled with personal data! measuring the logging of sensitive information in the android ecosystem," in *32nd USENIX Security Symposium*, 2023.

[14] F. Meneghello, M. Calore, D. Zucchetto, M. Polese, and A. Zanella, "Iot: Internet of threats? a survey of practical security vulnerabilities in real iot devices," *IEEE Internet of Things Journal*, 2019.

[15] D. Mauro Junior, L. Melo, H. Lu, M. d'Amorim, and A. Prakash, "A study of vulnerability analysis of popular smart devices through their companion apps," in *the IEEE Security and Privacy Workshops (SPW)*, 2019.

[16] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations," *IEEE Communications Surveys & Tutorials*, 2019.

[17] M. Kollat, "I tried the three most popular cheap fitness trackers on amazon. my verdict? you get what you pay for…," https://www.t3.com/features/i-tried-the-three-most-popular-cheap-fitness-trackers-on-amazon-here-is-my-honest-opinion.

[18] C. Aspinall, "The best heart rate monitors: track your vitals on the go," https://www.therecommended.com/fitness/best-heart-rate-monitors/.

[19] "Google play store for android - apk download," https://apkpure.com/google-play-store/com.android.vending.

[20] "App downloads for android - download, discover, share on uptodown," https://en.uptodown.com/.

[21] "A tool for reverse engineering 3rd party, closed, binary android apps." https://ibotpeaches.github.io/Apktool/.

[22] "skylot / jadx," https://github.com/skylot/jadx.

[24] P. Sivakumaran and J. Blasco, "A study of the feasibility of co-located app attacks against {BLE} and a {Large-Scale} analysis of the current {Application-Layer} security landscape," in *28th USENIX Security Symposium (USENIX Security)*, 2019.

[25] "Burp proxy - portswigger." https://portswigger.net/burp/documentation/desktop/tools/proxy.

[26] "Free apk downloads - free and safe android apk downloads," https://www.apkmirror.com/.

[27] "Log info disclosure | app quality | android developer," https://developer.android.com/privacy-and-security/risks/log-info-disclosure.