Identifying Channel Related Vulnerabilities in Zephyr Firmware

Devansh Rajgarhia
Indian Institute of Technology Kharagpur
Kharagpur, India
devanshrajgarhia@iitkgp.ac.in

Peng Liu
Pennsylvania State University
University Park, USA
pxl20@psu.edu

Shamik Sural
Indian Institute of Technology Kharagpur
Kharagpur, India
shamik@cse.iitkgp.ac.in

***Abstract*—In recent years, IoT devices and systems have helped make our lifestyle smarter. Operating systems running on IoT devices form a critical component for connectivity, security, networking, storage, remote device management and other system needs. As a result, applications deployed on top of such an operating system can exploit its vulnerabilities and potentially leak confidential data to the attacker. IoT devices typically have sensors that allow them to measure one or more channel values. They constitute one such example of confidential data for the user which can get leaked or manipulated by a malicious application exploiting the privileges provided by the operating system. In this work, we propose a methodology for finding security vulnerabilities using the concept of taint analysis on the LLVM IR of a part of the kernel of the Zephyr OS, a lightweight real-time operating system for connected, resource-constrained and embedded devices. Several vulnerabilities were detected as reported in the Results section.**

***Index Terms*—Vulnerability, Zephyr, LLVM IR, Taint Analysis**

I. Introduction

Internet of Things (IoT) is an evolving paradigm that allows electrical gadgets and sensors to communicate with each other through the Internet to make our lives easier. Smart devices and the Internet are used by IoT to bring new solutions to a variety of problems and concerns in commerce, administration and corporate entities throughout the world [1], [12]. Improvement in OS development is a critical step in creating a platform that supports the most up-to-date protocols and standards for intelligent IoTs of the future [13]. An IoT OS is expected to support a variety of hardware designs, boards and devices. RIOT [2], and Zephyr [3] are some of the IoT OSs that are available to help with the increasing growth in this sector.

The Zephyr operating system is based on a relatively simpler and smaller kernel. It is intended for use on asset constrained and embedded systems - from basic sensors to IoT remote applications running on smartwatches. Like other IoT operating systems, Zephyr also provides multiple drivers and sensor options to the user. The goal of this work is to analyze channel-related data leak vulnerabilities associated with the Zephyr RTOS. Zephyr channels are a relatively new concept in the embedded systems world. Fundamentally, a channel is a quantity that a sensor can measure. Since a complex sensor typically measures several quantities, the notion of channel enables an RTOS to have an abstraction (in the OS kernel) of individual quantities, especially when the sensor abstraction is too coarse-grained. It enables Zephyr to interact with a sensor in a fine-grained manner.

There are two kinds of vulnerabilities found in IoT OSs. First, the applications running on them may be malicious and could be leaking sensor channel values without the knowledge or consent of the user [11]. Even applications that are not malicious and were carefully programmed may suffer from such leaks (e.g., advertisements from third party). Second, it can also happen that an attacker takes advantage of the privileges given and manipulating the sensor channel data or the sensor device itself [14]. Denial of Service, Data Type Probing, Malicious Control, Malicious Operation, Scan, Spying, and Wrong Setup are examples of attacks and anomalies that can cause IoT system failures [16].

Taint analysis [15] is one of the methods used to detect such kind of security vulnerabilities in OSs. These approaches track sensitive "tainted" information through the OS by starting at a pre-defined source (e.g., an API method returning sensor channel value) and then following the data flow until it reaches a given sink (e.g., a socket), giving precise information about which data may be leaked where. Taint analysis can be used to identify the data that influences safety-critical components. In this work, we use the same approach to find security vulnerabilities present in the Zephyr OS kernel.

II. Related Work

Previously, there have been some work on analyzing frameworks and applications to find security vulnerabilities. Centaur is one such work that focuses on analyzing the Android Framework [7]. It enables symbolic execution of the framework and uses taint analysis to improve the effectiveness of vulnerability discovery. This analysis does not, however, include an analysis of any kind of channel-related vulnerabilities.

Gerbil is a firmware analysis specific extension of the Angr framework for analyzing binaries to effectively identify privilege separation vulnerabilities in IoT firmware [8]. It analyzes IoT firmware through symbolic execution and addresses the issue that an attacker may use the privilege separation vulnerability to perform a wide range of assaults, including malicious firmware replacement and denial of service. However, the privilege separation vulnerabilities are much different from the channel-related vulnerabilities that we have worked upon here.

TaintDroid, on the other hand, is a dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data [9]. It is used to monitor android applications and keep a check on how third-party applications use their private data. It employs taint analysis to label the private data of the user as sources and monitors in real-time how applications access and manipulate users' data. Flowdroid, in contrast, is a novel and exceptionally accurate static taint investigation tool for Android applications [10]. An exact model of Android's life cycle permits examination of callbacks conjured by the Android system. It works on the principle of forward and backward taint analysis to find the aliases of the tainted variable. Its objective is to detect private data leakage in Android applications, whereas in our work we detect sensor channel data leakage in an RTOS.

SCANDROID [18] is a tool for automatically reasoning about the security of Android apps. Its analysis is modular, allowing programs to be checked incrementally when they are installed on an Android device. It pulls security standards from manifests that come with such apps and verifies whether data flows through them are compliant with such specifications. It is a tool that does a data flow analysis but is only for Android apps whereas we here do it for an RTOS. CHEX [19] is a static analysis tool for automatically detecting component hijacking vulnerabilities in Android apps. It examines Android apps and discovers probable hijack-enabling flows by executing low-overhead reachability tests on customized system-dependent graphs, modeling such vulnerabilities from a data-flow analysis perspective.

FIRMADYNE [20] is an automated dynamic analysis solution that targets Linux-based firmware on network-connected commercial off-the-shelf devices. The design decisions solve several issues that come with dynamic analysis of COTS firmware. To accomplish the scale required to evaluate thousands of firmware binaries automatically, it relies on software-based full system emulation with an instrumented kernel.

Costin [21] assessed a collection of around 32,000 firmware images using static analysis. They found 38 previously undiscovered vulnerabilities, including hard-coded backdoors, embedded private key pairs, and XSS flaws, all of which were uncovered without undertaking advanced static analysis. Several alternative strategies for finding vulnerabilities in embedded devices have been developed to guard against this attack vector. For example, FIE [22] is a tool, which is a symbolic executor to discover vulnerabilities in embedded devices using the KLEE [23] symbolic execution engine. In a corpus of 99 open-source firmware applications for the MSP430 family of 8-bit embedded micro-controllers, they uncovered 21 memory safety vulnerabilities. At a lower level, FEMU [24] integrates the QEMU emulator inside the BIOS to simulate hardware peripherals during the development of an embedded SoC. However, none of the above-mentioned work focuses on static analysis of an RTOS and more importantly on detecting sensor channel-related vulnerabilities in an RTOS using taint analysis.

III. Detecting Vulnerabilities Using PhASAR

To perform the taint analysis, we use the LLVM [17] Intermediate Representation of a part of the kernel of the OS and process it. The analysis is often easier when a static analysis challenge is solved on the IR rather than the source language. This is because it eliminates the need for concrete source language, as the IR is often simpler due to the lack of nesting and fewer instructions. We use the lwm2m-client application code's prj.conf (Kernel configuration) file to first compile the app for getting the compile commands for each of the files that were being used for that particular project. In this way, the IR of only those components were generated that are necessary for our analysis, and most of the tasks that are not required were not integrated to avoid path explosion and the need to process huge exploded super graphs.

For our analysis, we use the PhASAR [4] static analysis tool. It is an LLVM-based static analysis framework for C/C++ code. PhASAR uses the Inter-procedural Distributive Environments algorithm to perform taint analysis on the IR. In IFDS [5] and its generalization IDE [6], a data flow problem is transformed into a graph reachability problem. Reachability is computed using the so-called exploded super-graph (ESG). The complexity of the IDE algorithm is $O(|N||D|^3)$, where $|N|$ is the number of nodes on the Inter-Procedural Control Flow Graph (or number of program statements) and $|D|$ is the size of the data-flow domain used. PhASAR requires the sources and sinks to be defined in a JSON file and feed this configuration for the analysis. PhASAR can also be used as a library to create an LLVM pass that runs on the LLVM IR for analysis. We made an LLVM pass using PhASAR to run the taint analysis on a defined set of sources and sinks.

A. Data Leakage

The first analysis done was to detect any kind of leakage of private data from the sensor to the Internet. For this purpose, we use our device as the source in the z_impl_sample_fetch and z_impl_channel_get functions, which are the system calls to get the channel values from the sensor connected to the IoT. We mark all the Internet-related functions from our LLVM IR as the sink and run the IDE Extended Taint Analysis of PhASAR. As expected, the tool shows that the source can reach the sink through some path in the IR although it is not capable of showing the exact path through which it is occurring. One of our speculated execution paths through which this can happen is by exploiting the APIs available for the application and using the CoAP protocol to transfer the resource data from the IoT device to the attacker. The time order of the APIs will be as follows. (a) sensor_sample_fetch: Fetches the sample value of a channel for a given device with the help of hardware and drivers. (b) sensor_channel_get: From the device's channel buffer, copies the fetched channel value and returns it to the user. (c) socket: Creates a socket for the transmission of data between a server and a client. (d) connect: Connects the sockets and initializes them. (e) coap_packet_init: Creates a new CoAP Packet from input data. (f) coap_packet_append_payload_marker: Append payload

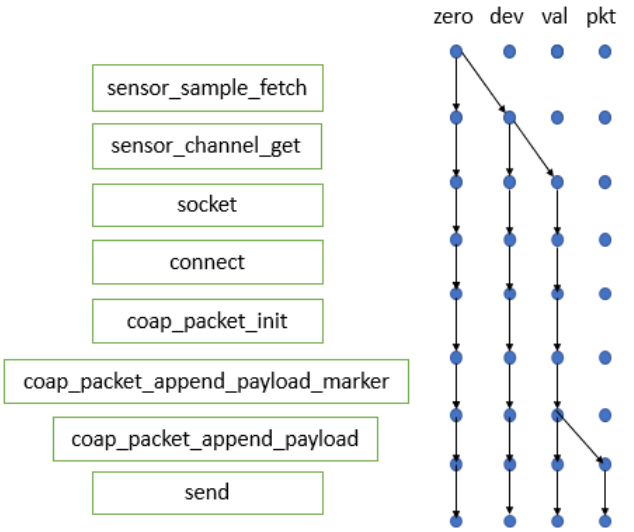

Fig. 1. Taint flow path for sensor channel leakage

marker to CoAP packet. (f) coap_packet_append_payload: Append payload to CoAP packet. (g) send: Sends the packet over the sockets.

In Fig. 1, we show how the data flow occurs when these APIs are called in that order. In the figure, *zero* refers to the tautology and is always tainted, *dev* refers to the sensor device being used to capture the sensor channel value, *val* refers to the sensor channel value returned by the sensor_channel_get API to the user and *pkt* refers to the network packet that will be transmitted across the internet. When the sensor_sample_fetch function is called, the device gets tainted with the sensor channel value. Once the sensor_channel_get function is called, the sensor channel data is copied from the device's buffer, thus tainting the *value* variable being returned to the user. Next, the user creates a socket, connects it to the other end of the socket using the address, and initializes a CoAP packet that is to be sent, using the coap_packet_init API. To add the sensor data as a payload to the packet, it uses the coap_packet_append_payload_marker and the coap_packet_append_payload APIs, respectively. During this process, the packet gets tainted due to the value. The packet is then sent to the other end of the socket using the send function, thus leaking the sensor channel values.

This is only one possible path. However, there could be several other possible paths through which this type of leakage can occur. Zephyr provides several other protocols like MQTT, LwM2M, HTTP, etc., which could be exploited by the attacker. This kind of data leakage might be used by agencies to collect data from users and use them for their analysis. For example, an agency might be keeping track of the temperature in which a person generally stays in, and accordingly shows advertisements for either heaters or air-conditioners. There is another type of vulnerability that is common in which an attacker can manipulate the sensor channel data.

B. Peripheral Manipulation

This analysis was done to see if the attacker can change the sensor channel values using the Internet. For this purpose, we use the z_impl_sample_fetch and z_impl_channel_get functions as the sink. We make an LLVM pass that uses PhASAR as a library. The pass could detect a declared variable in our LLVM IR and if the declared variable belongs to the internet-related codes, it would mark the variable as a source and run our taint analysis. In this way, we could get the variables that reaches our sinks by some path in the control flow graph. Using this LLVM pass, we were able to detect quite some vulnerabilities in the Zephyr OS.

Our analysis showed that the data and offset variables from the following functions when tainted, can reach the sinks (i.e., the sensor_channel_get or sensor_channel_fetch functions) and influence the sensors. (a) coap_packet_parse: Parses the CoAP packet in data, validating and initializing it. (b) parse_option: Parses the options of the CoAP packet. (c) decode_delta: The single-byte or two-byte length of the option is decoded. (d) read: Reads the data. (e) read_u8: Reads 8 bits of data. (f) read_be16: Reads 16 bits of data.

We can also confirm this taint propagation from the call-graph of the function coap_packet_parse shown in Fig. 2. The functions marked in red are those involved in the propagation of the tainted data and offsets. The callgraph shows that these tainted variables are passed onto the parse_option function, which then passes the tainted variables to read_u8, decode_delta, and read functions. The decode_delta function passes these tainted variables to read_be16.

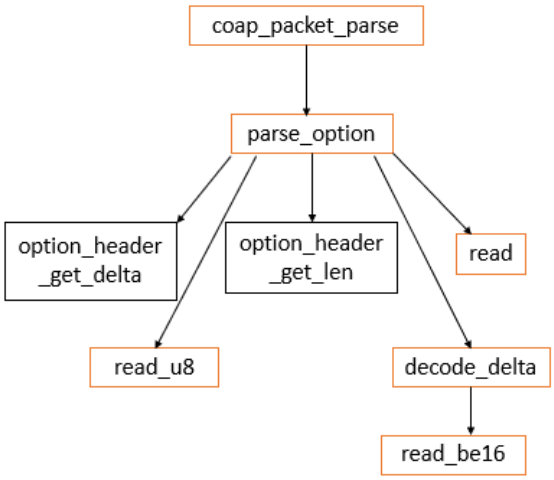

Fig. 2. Call graph for coap_packet_parse function

This data variable in the true sense is the data containing a CoAP packet, its data pointer being positioned at the start of the packet. There are several ways how this can be used by an attacker. One is by designing an application with the following time order execution of the APIs. (a) socket: Creates a socket for the transmission of data between a server and a client. (b) connect: Connects the sockets and initializes them. (c) recv:

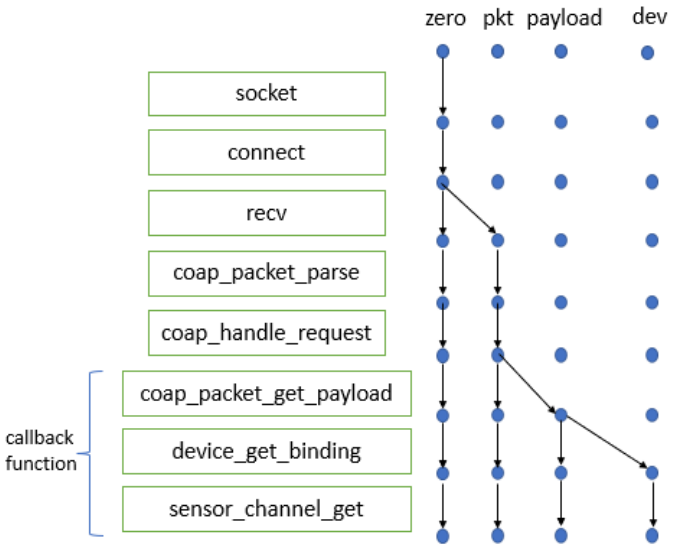

Fig. 3. Taint flow path for sensor channel manipulation

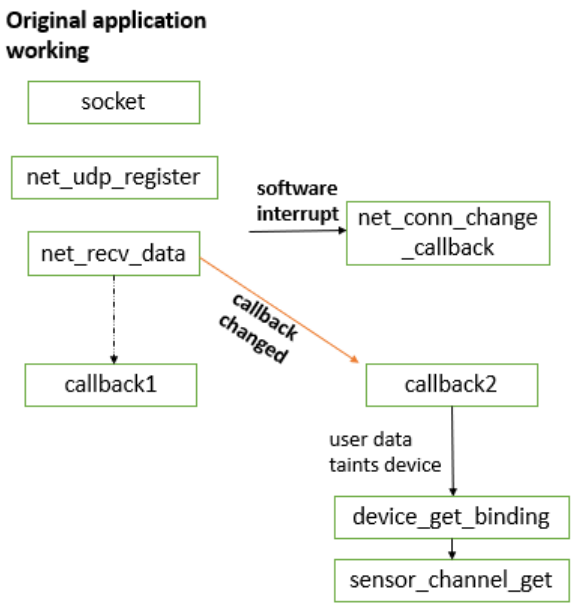

Fig. 4. Vulnerability in case of net_conn_change_callback

Recieve the packet of the socket. (d) coap_packet_init: Creates a new CoAP Packet from input data. (e) coap_packet_parse: Parses the CoAP packet in data, validating and initializing it. (f) coap_handle_request: When a request is received, calls the appropriate methods of the matching resources. (g) coap_packet_get_payload: Returns the data pointer and length of the CoAP packet. (h) sensor_channel_get: From the device's channel buffer copies the fetched channel value and returns it to the user.

The taint flow for this kind of application is shown in Fig. 3. In the designed application, the IoT device first creates a socket, connects to the address of the attacker, and receives the tainted packets through the Internet. It then parses the packet using the coap_packet_parse function and after validation, calls the coap_handle_request function. The coap_handle_request function calls the appropriate callback function, in which it can get the payload from the packet using the coap_get_payload function and then use this payload to get access to a sensor in the IoT device. Once the attacker has access to the device, he can call the sensor_channel_fetch or any other related APIs to manipulate the data in the device buffer. It can also call for sensor_attr_set API to change the attributes like range, sampling frequency, or any other device configuration which is supported for that particular sensor. This can also result in the crashing of the IoT device due to incorrect API calls for the sensor.

The PhASAR tool also detects a vulnerability in the net_conn_change_callback function. This function changes the callback and user_data for a registered connection handle. If the callback is changed and set to some other callback that can access the sensor device, it can cause a problem. Consider the example path shown in Fig. 4. The flow is as follows. (a) socket: Creating a socket on the IoT device. (b) net_udp_register: Registering a callback to be called when a UDP packet is received corresponding to the received packet. (c) net_recv_data: Pushing the packet up in the network stack for further processing. (d) callback1: Executing Callback when the UDP packet is received.

The application first opens a socket and connects to the address of the other socket. Once connected, it registers a callback for UDP packets using a protocol for the connection (i.e., UDP here), protocol family-like AF_INET6 for IPv6 support, and the socket addresses of the endpoints. Every time a packet arrives the callback1 function is called and executed. This application runs normally until a software interrupt is made. If this software interrupt calls for net_conn_change_callback, the callback function for this application can be changed to callback2 and the user data is also changed to the new user data that has been provided by the software interrupt. This user data can be used by the new callback2 function to get hold of a sensor using the device_get_binding function. Callback2 can then also call sensor channel-related functions for the particular sensor. The taint flows from user data supplied by the net_conn_change function call to the sensor device.

Another vulnerability that arises is due to the net_conn_input function. It is called when a network packet is received by the IoT device. It returns a verdict NET_OK if the packet was consumed or NET_DROP if the packet parsing failed and the caller should handle the received packet. If the packet was consumed, it means that it called a callback function for that packet and in the callback function, it could access the device/sensor of the IoT device similar to the way described above. We show a partial callgraph of net_conn_input function in Fig. 5 and its taint propagation along the edges. The functions marked in red are the ones detected by PhASAR as a vulnerability. If we assume the

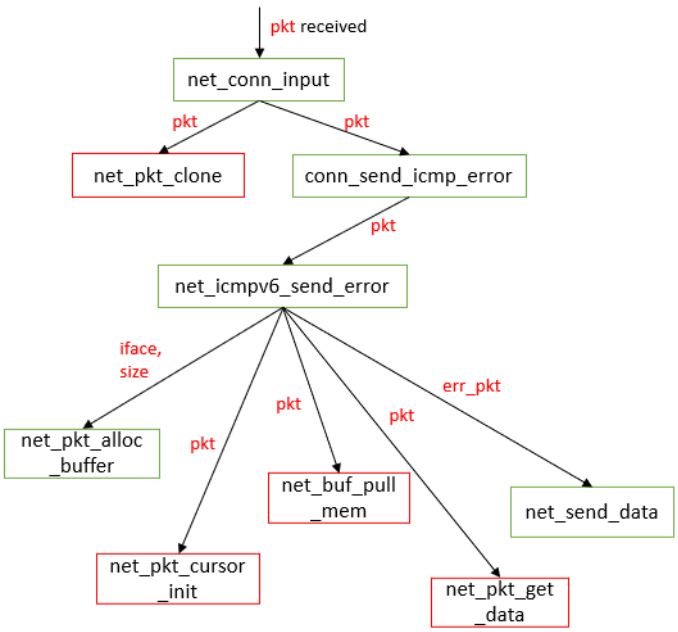

Fig. 5. Call graph of net_conn_input and taint propagation

packet received (denoted by pkt in Fig. 5) to be tainted, then with the execution of code, it taints more variables along the path of the callgraph, which are also detected by PhASAR. The function checks if it received a packet with a multicast destination address, since then it might need to deliver the packet to multiple recipients. In this case, it calls net_pkt_clone to make a clone of the packet. The clone here is detected as a vulnerability by PhASAR.

It can also be seen from the callgraph that the net_conn_input function sends an ICMP error using the conn_send_icmp_error function and passes the tainted packet to it. The packet is passed to net_icmpv6_send_error function. This function makes a new packet (denoted by err_pkt in Fig. 5) using the net_pkt_alloc_with_buffer function. It initializes the cursor using the net_cursor_init function and reads the headers of the original packet. The source and destination link addresses from the original packet are copied to this new packet. It uses the net_buf_pull_mem function to decode the data in the buffer and set the lengths of the copied packet. The err_pkt is then sent using the net_send_data function. The taint further propagates into the net_pkt_alloc_buffer function. It finally reaches pool_get_unint, data_alloc and net_buf_frag_insert functions. If any of these tainted variables is changed by the attacker, it affects the packet that was received and thus, it also affects the callback function which is executed after receiving the packet.

IV. Experimental Results

The analysis was done on a virtual machine having 16 VCPUs (Intel Xeon processors) and 32 GB RAM.

PhASAR v0521 and Zephyr v2.7.1 were used for the analysis. On the VM, it took around 20 minutes for the LLVM pass to run on the IR for the peripheral manipulation analysis. This time is quite reasonable since the IR has 388,499 instructions resulting in a huge control flow graph. Also, the IDE Extended Taint Analysis of PhASAR was run 1,855 times. This analysis time could be reduced if instead of doing a forward taint analysis, we did a backward taint analysis.

TABLE I
Vulnerabilities found by peripheral manipulation analysis

File Name	Function Name	Variables
lwm2m_engine.c	lwm2m_init	_s_buffer
lwm2m_rw_json.c	get_s32	tmp
	get_objlnk	tmp, value_offset, fd
lwm2m_rw_oma_tlv.c	put_end_oi	out, fd
	put_end_ri	out, path, fd
	put_s8	out
	put_s16	out, value
	put_s32	out, value
	put_s64	out, value
	put_string	out, buflen
	put_float	out
	put_bool	out, value_s8
	put_opaque	out, buflen
	put_objlnk	out
	get_s32	temp, tmp
lwm2m_rw_plain_text.c	get_opaque	in, opaque, in_len
	get_objlnk	tmp
buf.c	fixed_data_alloc	size, fixed
	pool_get_uninit	pool
	data_alloc	size
	net_buf_get	ret
	net_buf_simple_reserve	buf
	net_buf_clone	buf, clone, size
	net_buf_frag_insert	parent, frag
	net_buf_simple_push	buf
	net_buf_simple_pull	buf
	net_buf_simple_pull_mem	buf
connection.c	conn_get_unused	node, cb, user_data
net_pkt.c	net_pkt_clone	backup
	net_pkt_cursor_init	pkt
	net_pkt_frag_insert	pkt, frag
	net_pkt_get_data	backup
	net_calc_chksum	backup
coap.c	coap_packet_parse	opt_len
	parse_option	opt_delta
	read_u8	data, offset
	decode_delta	data, offset, opt
	read	offset
	read_be16	offset
	coap_block_transfer_init	block_size, total_size
	coap_update_from_block	size1, size2
	update_control_block2	ctx, new_current
	update_control_block1	ctx, size
	update_descriptive_block	ctx, size, new_current
	coap_reply_init	tkl

However, since the time taken by PhASAR for the analysis is only 20 minutes, it was felt to be sufficient to go with the forward analysis. Out of these 1,855 analyzed variables, we could detect vulnerabilities in 75 variables for the peripheral manipulation analysis. The detected vulnerabilities are listed in Table I. Further examination revealed that most of these

variables are data packets received from the Internet, and then that data is used in some callback function. Using the callback function along with the data in the packet, the attacker can manipulate the sensor device or any related sensor channel as explained in the previous section. Some of the detected vulnerabilities are also straightforward like the functions which can directly manipulate sensor values. For example, LwM2M payload can be formatted as TLV (Type-Length-Value), JSON (JavaScript Object Notation), opaque or Plain Text, and all such formats are supported by the Zephyr OS. PhASAR also reports these functions, which are used for text formatting in LwM2M protocol as not secure. Some of these are (a) put_s8: Set resource to value (signed 8 bit integer) (b) put_s32: Set resource to value (signed 32 bit integer) (c) put_string: Set resource to value (string) (d) put_bool: Set resource to value (boolean)

Each of these functions is used to set a resource value in a particular format. For example, the above functions are used to set the resource's value in TLV format. These functions can be used by the attacker to set the sensor values using the LwM2M protocol. For the data leak analysis, it takes around 3 minutes for our LLVM pass to run on the IR. Since here we made all the sources together and ran the IDE Extended Taint Analysis just once, it takes much less time than the peripheral manipulation analysis. The majority of the time our analysis is utilized by PhASAR for building the control flow graph.

V. Conclusions and Future Directions

We have presented a methodology for static analysis of Zephyr Firmware to find sensor channel-related vulnerabilities. We have been able to identify two kinds of vulnerabilities related to the sensor channel, the first being data leakage from the sensor to the Internet and the second being sensor data manipulation using the Internet. We have provided several sample paths the attacker can take advantage of to disrupt the activities of the user or to gather information from the user without her content. Both types of vulnerabilities need to be addressed urgently.

To further confirm these paths through which data leakage or data manipulation is taking place, we need to look into the control flow graph of the IR. We plan to do this in our future work. Using the control flow graph, we can find the exact paths through which these bugs can be exploited. We also plan to do static analysis of the rest of the kernel to discover more vulnerabilities in the Zephyr OS, which might not be related to the sensor channel but some other peripheral.

Acknowledgement

Peng Liu was supported by NSF CNS-1814679 and NSF CNS-2019340.

References

[1] Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M.S. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems. 1645-1660.

[2] RIOT : The Friendly Operating System for Internet of Things. Available online :https://www.riot-os.org/

[3] Zephyr Project. Available online: https://www.zephyrproject.org/

[4] Schubert, P., Hermann, B., & Bodden, E. (2019). PhASAR: An Inter-procedural Static Analysis Framework for C/C++. TACAS, 393-410.

[5] Reps, T., Horwitz, S., & Sagiv, S. (1995). Precise interprocedural dataflow analysis via graph reachability. POPL '95, 49-61.

[6] Sagiv, S., Reps, T., & Horwitz, S. (1995). Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. Theoretical Computer Science - TCS, 167, 651-665.

[7] Luo, L., Zeng, Q., Cao, C., Chen, K., Liu, J., Liu, L., Gao, N., Yang, M., Xing, X., & Liu, P. (2020). Tainting-Assisted and Context-Migrated Symbolic Execution of Android Framework for Vulnerability Discovery and Exploit Generation. IEEE Transactions on Mobile Computing, 19, 2946-2964.

[8] Yao, Y., Zhou, W., Jia, Y., Zhu, L., Liu, P., & Zhang, Y. (2019). Identifying Privilege Separation Vulnerabilities in IoT Firmware with Symbolic Execution. ESORICS, 638-657.

[9] Enck, W., Gilbert, P., Chun, B., Cox, L.P., Jung, J., Mcdaniel, P., & Sheth, A. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Trans. Comput. Syst., 32, 5:1-5:29.

[10] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., & Mcdaniel, P. (2014). FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 259–269..

[11] Park, M., Oh, H., & Lee, K. (2019). Security Risk Measurement for Information Leakage in IoT-Based Smart Homes from a Situational Awareness Perspective. Sensors (Basel, Switzerland), 19, 2148.

[12] Kumar, S., Tiwari, P., & Zymbler, M.L. (2019). Internet of Things is a revolutionary approach for future technology enhancement: A review. Journal of Big Data, 6, 1-21.

[13] Baccelli, E., Hahm, O., Günes, M., & Wählisch, M. (2013). Operating Systems for the IoT – Goals , Challenges and Solutions. Proceedings of WISG.

[14] Liu, X., Liu, Y., Liu, A., & Yang, L.T. (2018). Defending ON–OFF Attacks Using Light Probing Messages in Smart Sensors for Industrial Communication Systems. IEEE Transactions on Industrial Informatics, 14, 3801-3811.

[15] Feng, Z., Wang, Z., Dong, W., & Chang, R. (2018). Bintaint: A Static Taint Analysis Method for Binary Vulnerability Mining. 2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB), 1-8.

[16] Sikder, A.K., Petracca, G., Aksu, H., Jaeger, T., & Uluagac, A.S. (2021). A Survey on Sensor-Based Threats and Attacks to Smart Devices and Applications. IEEE Communications Surveys & Tutorials, 23, 1125-1159.

[17] Lattner, C., & Adve, V.S. (2004). LLVM: a compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004. CGO 2004., 75-86.

[18] Fuchs, A.P., Chaudhuri, A., & Foster, J.S. (2009). SCanDroid : Automated Security Certification of Android Applications.

[19] Lu, L., Li, Z., Wu, Z., Lee, W., & Jiang, G. (2012). CHEX: statically vetting Android apps for component hijacking vulnerabilities. Proceedings of the 2012 ACM Conference on Computer and communications security, 229-240.

[20] Chen, D.D., Woo, M., Brumley, D., & Egele, M. (2016). Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. NDSS.

[21] Costin, A., Zaddach, J., Francillon, A., & Balzarotti, D. (2014). A Large-Scale Analysis of the Security of Embedded Firmwares. USENIX Security Symposium.In Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14), 95–110.

[22] Davidson, D., Moench, B., Ristenpart, T., & Jha, S. (2013). FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. USENIX Security Symposium, 463-478.

[23] Cadar, C., Dunbar, D., & Engler, D.R. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI, 209-224.

[24] Li, H., Tong, D., Huang, K., & Cheng, X. (2010). FEMU: A firmware-based emulation framework for SoC verification. 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 257-266.