SEESAW: A Tool for Detecting Memory Vulnerabilities in Protocol Stack Implementations

Farhaan Fowze
University of Florida
USA
farhaan104@ufl.edu

Tuba Yavuz
University of Florida
USA
tuba@ece.ufl.edu

Abstract

As the number of Internet of Things (IoT) devices proliferate, an in-depth understanding of the IoT attack surface has become quintessential for dealing with the security and reliability risks. IoT devices and components execute implementations of various communication protocols. Vulnerabilities in the protocol stack implementations form an important part of the IoT attack surface. Therefore, finding memory errors in such implementations is essential for improving the IoT security and reliability. This paper presents a tool, SEESAW, that is built on top of a static analysis tool and a symbolic execution engine to achieve scalable analysis of protocol stack implementations. SEESAW leverages the API model of the analyzed code base to perform component-level analysis. SEESAW has been applied to the USB and Bluetooth modules within the Linux kernel. SEESAW can reproduce known memory vulnerabilities in a more scalable way compared to baseline symbolic execution.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software and application security**;

Keywords: IoT, Bluetooth, USB, memory vulnerability, symbolic execution, static analysis

ACM Reference Format:
Farhaan Fowze and Tuba Yavuz. 2021. SEESAW: A Tool for Detecting Memory Vulnerabilities in Protocol Stack Implementations. In *19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '21), November 20–22, 2021, Beijing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3487212.3487345

MEMOCODE '21, November 20–22, 2021, Beijing, China

ACM ISBN 978-1-4503-9127-6/21/11…$15.00
https://doi.org/10.1145/3487212.3487345

1 Introduction

The landscape of Internet of Things (IoT) is vast and diverse. A typical IoT ecosystem consists of constrained IoT devices, routers/edge devices, mobile phones, and the cloud. The communication protocol implementations form an important part of the attack surface due to their complexity. The Linux kernel forms an important of the IoT landscape due to its market share on routers, edge devices, and mobile phones. The complexity of the APIs in the Linux kernel makes it challenging to analyze and identify vulnerabilities in the implementations of communication protocols such as the USB and Bluetooth protocols.

A recent report on the Blueborne family of vulnerabilities [1] that were found in various implementations of the Bluetooth protocol indicate the difficulty of using dynamic analysis for detecting vulnerabilities. Therefore, static program analysis approaches that do not require a real device offer advantages over dynamic analysis. Static analysis allows one to achieve high coverage. However, it is difficult to strike a balance between precision and analysis. Dynamic symbolic execution, as implemented in KLEE [5], provides a precise memory model, but it has limited scalability due to the path explosion problem.

In this paper, we introduce a tool, called SEESAW[1], that combines the scalability of static analysis with the precise memory model of symbolic execution to detect memory vulnerabilities in IoT relevant protocol stack implementations. SEESAW applies symbolic execution at the component level and under the guidance of static analysis, which can provide a more complete view of the program state space. Our approach leverages Directed Symbolic Execution (DSE) [10], which is a well-known technique for guiding symbolic execution to a target code location. However, our approach differs from previous applications of DSE in three ways: 1) SEESAW focuses on detecting memory errors that are protocol-relevant, which are induced due to wrong handling of the protocol fields, instead of generic errors. SEESAW uses a novel approach to automatically detect protocol relevant program variables. 2) SEESAW utilizes under-constrained symbolic execution [7, 11] to perform modular execution on the analyzed programs. 3) SEESAW allows two ways of communication between the symbolic execution and the

[1]Source code is available at https://drive.google.com/drive/folders/12Og3-zRjNoYEjJBJHiFRwNMl2c4y664-?usp=sharing

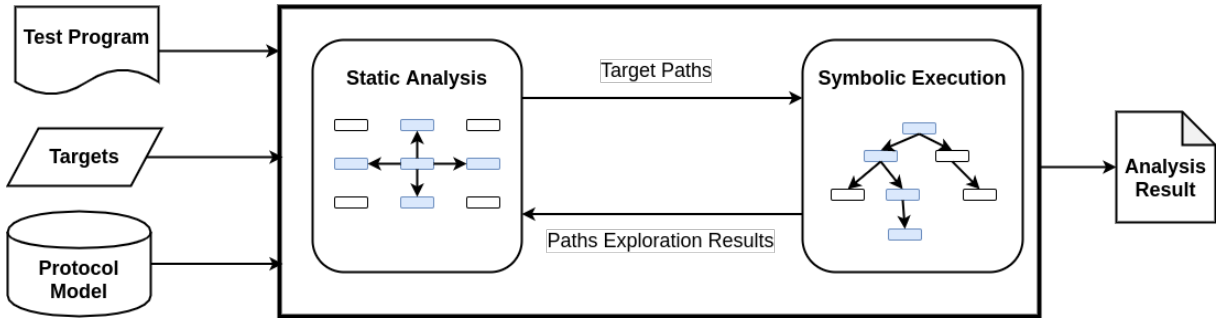

Figure 1. Architecture of SEESAW for protocol guided vulnerability detection.

static analysis components. While static analysis provides target paths to symbolic execution, guided symbolic execution provides the static analysis with resolved function pointer information to improve the precision and completeness of the analysis. We have applied our approach to the widely used USB and Bluetooth modules from the Linux kernel.

The paper is organized as follows. Section 2 provides an overview of our approach Section 3 presents our results on applying SEESAW to the USB and Bluetooth implementations in the Linux kernel. Section 4 discusses related work. Section 5 concludes with directions for future work.

2 Overview

This section presents the overview of SEESAW, our protocol guided vulnerability detection methodology. We utilize protocol information in detecting vulnerabilities within protocol stack implementations. Figure 1 shows the major components of SEESAW.

2.1 Specifying and Mapping Targets

Our analysis engine finds out targets based on protocol domain information. Our analysis engine needs protocol dependent aspects to be specified. We can define the types of the protocol relevant fields as input. Most of these domain relevant fields are defined within some special data type, e.g., structs in the Linux kernel code. We can specify these at the file level. For each protocol we want to model, we can specify the list of header files that contain the protocol relevant fields. This is a reasonable assumption since the protocol stack implementations are reasonably structured and modular. Based on the protocol information, the analysis engine first maps the targets within the program. For example, to detect memory corruption errors we specified targets as `memcpy` callsites. The target mapper initially finds out all explicit calls to `memcpy`. This is then passed to our core execution engine shown in Figure 1. The targets are systematically filtered in the next stages using domain informed analysis. Our system works on the LLVM IR. Clang compiler generated bitcode is traversed to find the required target instructions.

2.2 Static Analysis Component

SEESAW utilizes an open-source static analysis tool SVF [3] for static analysis. SVF is the state-of-the-art tool for inter-procedural value-flow and pointer analysis of C programs. It is scalable, and built on the LLVM IR. SVF can build value-flow information and points-to analysis in an iterative manner. We utilize SVF's ability to integrate external points-to analysis results to improve precision. SVF divides pointer analysis into loosely coupled Graph, Rules and Solver [12]. These three components allows users to tailor the tool according to their analysis needs. The recent advancement of the tool allows customizable analysis: context, flow, and field sensitive pointer analysis.

Once the targets are mapped within the bitcode, we utilize SVF's value-flow graph to gain insights on the targets. We treat the targets as sinks and try to explore the inter-procedural value-flow to find out all the sources from which the target gets its values. We use the context-sensitive analysis provided by SVF. Our extension here is two-fold. Firstly, we utilize our protocol guidance in the analysis. Using the domain information SEESAW finds out the targets that are dependent on protocol relevant fields. For these targets we find out all the incoming paths from different function entry points. These paths are passed to the symbolic execution component along with the entry points. The second extension is related to the interaction with the symbolic execution component. We utilize our symbolic execution component in resolving function pointer targets that static analysis could not resolve. This resolution is passed back to the static analysis component to get more precise value-flows and control dependencies. Therefore, our static analysis is multi-pass. It interacts with the symbolic execution resolutions until a fixed point is reached in the function pointer target resolutions.

2.3 Symbolic Execution Component

The symbolic execution component of SEESAW is built on the PROMPT tool [13], which implements under-constraint symbolic execution and API model guided analysis on top of KLEE [5]. PROMPT allows specification of an environment in the form of API models. Specifically, we have specified the enclosing relationship between struct types to deal with the pointer arithmetic used in the `container_of` macro that is heavily used in the Linux kernel. Since such pointer arithmetic uses a negative offset, it represents a challenge for static analysis tools like SVF, which assume that the pointer arithmetic expressions do not go outside of the object.

Our approach extends the standard symbolic execution in KLEE with a path prioritization algorithm that prefers paths that follow the sequence of basic blocks as specified by the static analysis over those that deviate. The idea is to spend the resources on paths that are likely to reach target locations or protocol relevant memory access locations. For a given set of target paths, our symbolic execution engine performs two tasks. First, it explores the path to target starting from the given entry point. This is the detection phase where we try to figure out if there exist any out of bound memory accesses in the path. Any error condition found during the exploration is recorded. Second, we record any function pointer resolutions during the path exploration. The resolutions may come from following the exact path or some paths feasible from the entry point. This result is passed back to the static analysis engine so that it can leverage this information. We employ caching in the solver provided by the symbolic execution engine to reduce the time in solving queries.

2.4 Combining Static Analysis and Symbolic Execution

This work combines two major analysis techniques: static analysis and symbolic execution. This combination allows us to gain from the qualities of both. Static analysis is scalable but imprecise. Symbolic execution utilizes a precise memory model but runs into the path explosion problem. SEESAW's static analysis engine provides the protocol-relevant targets to the symbolic execution engine to explore. The symbolic execution engine explores the paths to find bugs and to record function pointer resolutions. The new found function pointer resolutions are provided back to static analysis which can explore more parts of the code. The combination iteratively finds and explores all detected targets. To keep the components independent we use source line information as the basis of communication between the engines. We chose to use source information as the basis so that the engines can perform regardless of the tooling issues, e.g., the version of the IR. Our approach works on the LLVM IR. The target paths from static analysis are first recorded as sequences of basic blocks. For each path, a summary is created using the terminating instruction of the basic blocks in the path. This summarized sequence is used by symbolic execution in determining for each path which successor to prioritize.

Our hybrid exploration heuristic can be performed in various configurations. For example, symbolic execution can be started from any specified entry point to reason about a known target. If no memory errors are found, it might very well be the case that static analysis is not required further. In the case of detecting a memory error, there is a possibility that it is a false positive due to the under-constrained context. So, static analysis can be used to find a specific calling context and to provide the paths with constrained contexts back to symbolic execution so that the bugs within that context can be confirmed. As we have seen in many experiments the constraint context reduces these false positives. The complex pointer arithmetic that could not be handled by static analysis gets resolved by symbolic execution and the path explosion in symbolic execution stays contained due to prioritizing relevant paths provided by static analysis.

The example code snippet provided in Figure 2 is from the Linux kernel version 4.14-rc2. It is a part of the Bluetooth module. We provided `memcpy` functions as targets to find out memory safety problems. The target mapping phase of our analysis first identified a target, the `memcpy` function called on line 35. Through static analysis we were able to find out all the possible paths to this function call. The analysis engine then found the sources of the parameters. The destination buffer, *scan_rsp_data* field of *adv_instance*, and the length of the copy operation, *scan_rsp_len* field of *adv_instance*, were found to be dependent on some protocol fields. This is because *adv_instance* is a pointer of type "*struct adv_info*", which is defined in the file "*hci_core.h*". SEESAW uses the header file paths related to a protocol implementation for modeling protocol information. Since the header file "*hci_core.h*" is stored under one such path, `include/net/bluetooth`, "*struct adv_info*" is considered protocol relevant, and, hence, the values used as the destination buffer and length parameters to the `memcpy` at line 35 are also considered protocol relevant. The three possible entry points for this `memcpy` callsite include the functions *create_instance_scan_rsp_data*, *set_local_name*, and *__hci_req_update_scan_rsp_data*. Symbolic execution engine is then passed all these paths grouped by the entry point to explore on. The symbolic execution can now start from each of the entry point listed above and search for any memory vulnerabilities involving the targets.

3 Evaluation

To evaluate the effectiveness of SEESAW, we applied it to the detection of protocol relevant vulnerabilities in the Linux kernel, specifically, to the Bluetooth and USBCore modules. The version we utilized was v4.14-rc2. We used the clang compiler to generate the bitcodes. We utilized two versions of the clang compiler due to tooling issues. The static analysis

```
1  // net/bluetooth/mgmt.c:3083
2  static int set_local_name(struct sock *sk,
3       struct hci_dev *hdev, void *data, u16 len) {
4    ...
5    if (!memcmp(hdev->dev_name, cp->name,
     ↪  sizeof(hdev->dev_name)) &&
6        !memcmp(hdev->short_name, cp->short_name,
         ↪  sizeof(hdev->short_name))) {
7          ...
8          goto failed;
9    }
10   ...   // More conditions are checked
11
12   if (lmp_le_capable(hdev) && hci_dev_test_flag(hdev,
     ↪  HCI_ADVERTISING))
13        __hci_req_update_scan_rsp_data(&req,
          ↪  hdev->cur_adv_instance);
14        ...
15 }
16
17 // net/bluetooth/hci_request.c:1055
18 void __hci_req_update_scan_rsp_data(struct
19 hci_request *req, u8 instance) {
20   struct hci_dev *hdev = req->hdev;
21   struct hci_cp_le_set_scan_rsp_data cp;
22   ...
23   if (instance)
24     len = create_instance_scan_rsp_data(hdev, instance,
       ↪  cp.data);
25   ...
26 }
27
28 // net/bluetooth/hci_request.c:1027
29 static u8 create_instance_scan_rsp_data(struct
30 hci_dev *hdev, u8 instance, u8 *ptr) {
31   struct adv_info *adv_instance;
32   u32 instance_flags;
33   u8 scan_rsp_len = 0;
34   ...
35   memcpy(&ptr[scan_rsp_len],
     ↪  adv_instance->scan_rsp_data,
36   adv_instance->scan_rsp_len);
37   ...
38 }
39
40 // include/net/bluetooth/hci_core.h:198
41 struct adv_info {
42   ...
43   __u16  scan_rsp_len;
44   __u8  scan_rsp_data[HCI_MAX_AD_LENGTH];
45 };
```

Figure 2. A minimal code snippet of a Bluetooth protocol within the Linux kernel with a protocol relevant target at line 35.

component utilized clang-7 to generate the LLVM bitcodes. The symbolic execution component utilized clang-3.8, which was the most recent LLVM version in PROMPT, and, hence, in our underlying symbolic execution tool KLEE. The basis of information sharing between the two analysis components is the source line information. So, all the modules were built with the debug flag to enable source line recordings for each instruction in the LLVM bitcode. Section 3.1 evaluates the effectiveness of our protocol guided target filtering and Section 3.2 evaluates the performance of our guided symbolic execution component. Section 3.3 evaluates the effectiveness of SEESAW in terms of bug detection.

3.1 Protocol Relevant Target Finding

The static analysis phase is responsible for protocol relevant target finding. We have generated the protocol knowledge for Bluetooth from the header files under */include/net/bluetooth/* and */net/bluetooth/* directories. For USB, we used the header files from */include/linux/usb.h*, */drivers/usb/core/*, and */include/uapi/linux/usb/ch9.h*. We consider the structures and their fields that were defined in these files to be protocol relevant. Our protocol knowledge informed static analysis achieved great reductions in the number of paths and the number of targets to test. As Table 1 shows, we found 461 and 35 protocol relevant fields in the Bluetooth and USB modules, respectively. There were a total of 397 (Bluetooth) and 20 (USB) memcpy targets in the modules. Using the protocol information, we were able to reduce the number of targets to 203 for Bluetooth and only to 4 for USB. This allowed us to gain 48.87% reduction in the Bluetooth targets and 40% reduction in the USB targets.

Ground truth evaluation. To evaluate the effectiveness of our API model based approach, we have analyzed all the targets in the USB and the Bluetooth modules. Using our static analysis engine we have found all the struct types the targets are dependent on. Based on the list of files provided earlier, we have determined the protocol relevance of each of these structs. All the protocol relevant targets we have found were precise for both modules. For example, 28 out of the 39 protocol relevant types in the USB header files are the protocol fields/descriptors like *device_descriptor*, *config_descriptor* etc. The rest of the types includes struct types that represent various abstractions in the protocol like *usb_host_endpoint*, *usb_host_config*, etc. For the Bluetooth module, there were 227 struct types like *hci_command_hdr* and *hci_event_hdr* related to the Host Controller Interface (HCI) layer. There were 36 types including *l2cap_cmd_hdr* and *l2cap_conn_req* related to the Logical Link Control and Adaptation Layer Protocol (L2CAP) layer. The rest of the types include implementations from the Bluetooth modules smp, amp, mgmt, bnep, and rfcomm.

As shown in Table 1, we have achieved great reductions in analysis targets by determining the protocol relevant targets. For USBCore we found that 8 of the targets were not dependent on the USB protocol. We found these targets to be dependent on core kernel relevant structs like *kernel_symbol* and *kernfs_node*. For the Bluetooth module we have achieved higher reduction in the number of

Table 1. Protocol relevant target finding using static analysis

Protocol	#Relevant fields	#Targets	#Protocol relevant targets	Reduction
Bluetooth	461	397	203	48.87%
USB	39	20	12	40%

targets. From the sources of the reduced targets, we have seen that our approach missed two protocol relevant types: *smp_chan* and *sco_pinfo*. These struct types were defined in */net/bluetooth/sco.c* and */net/bluetooth/smp.c*. Since our API model focused on structs defined in the header files, the approach could not mark these structs as protocol relevant.

3.2 Performance of Guided Symbolic Execution

Based on the target paths provided by static analysis, our symbolic execution explores relevant portions of the program to gain coverage. Compared to the baseline symbolic execution, our guided execution covers relevant code with significant speed-up. A representative set of Bluetooth target timing data is shown in Table 2. There are multiple entry points for some of the targets. Our symbolic execution step runs on groups of paths that start from the same entry point and end at the same target. This allows us to build a comprehensive execution context on each target. We ran our symbolic execution phase with a timeout of 6 hours. The maximum time taken for our prioritized execution is 875.43 seconds for Target No. 3 as shown in Table 2. Baseline symbolic execution could not reach this target within the given time bound. There were 32-41 branches in different paths to this target. Baseline symbolic execution faced path explosion before it could follow the relevant paths. The target that took the least time is Target No. 2. It took 1.8 seconds for SEESAW to explore compared to 2.4 seconds on the baseline. Similarly, targets 8-10 have almost similar performance for baseline and SEESAW. The reason is that for these paths there were only a few branches to explore in the path. For such small code samples the baseline approach can cover the paths easily. For targets 3-7 and 11, baseline symbolic execution could not reach the targets. These paths were relatively longer and had multiple function calls with multiple loops. Therefore, it is difficult for the baseline symbolic execution to reach these targets.

Our approach has shown its effectiveness in exploring USB module related targets as well. None of the targets were reachable by the baseline symbolic execution engine. Our targets were reachable within the given time bound and we were able to execute all paths to the targets. Baseline symbolic execution could not scale in this regard, the USBCore module code size (18,004 lines of C) is smaller than the Bluetooth module (43,607 lines of C)[2]. Due to the high number of branches from the entry points, it is difficult for baseline symbolic execution to reach a target. In addition to that, all the USB targets were deep in the code. Baseline symbolic execution was not able to reach deep Bluetooth targets either.

Relevant code coverage. We have evaluated our relevant code coverage compared to the baseline symbolic execution to evaluate our approach. Our path prioritization technique is able to focus the execution directly on the required path. And even when the execution deviates the system can recognize the paths that return to the desired track and prioritize them accordingly. Our approach has achieved 100% coverage of the relevant code for all the targets. Baseline symbolic execution was able to achieve 100% coverage for the targets it could reach (Target 1, 2, 8-10). For the rest, the highest relevant code coverage of baseline symbolic execution is less than 10% within the given time bound. This is one of the reasons why path prioritization is necessary for analysis and effectively shows the impact of our analysis.

Pointer resolution. Our symbolic execution engine was able to resolve pointers that could not be resolved by static analysis for two of the entry points. This was enabled by API modeling of the `container_of` macro, which got involved in some function pointer expressions. Once resolved, we supplied the data back to static analysis engine. The resolved targets of the function pointers were small functions with minimal operations and did not contribute to the control-flow. Therefore, static analysis did not find any new paths spawning from them.

3.3 Bug Detection

We have evaluated SEESAW's bug detection capability by applying it to the detection of known bugs in both the USBCore and the Bluetooth modules. Our results have shown that SEESAW can effectively detect memory vulnerabilities in both modules.

Bluetooth vulnerability detection. We have applied SEESAW to detecting one (CVE-2017-1000251 [2]) of the BlueBorne [1] family of vulnerabilities. The vulnerability is manifested in the extended flow specification (EFS) feature of the L2CAP protocol implementation in the Linux kernel. There was a lack of size check in the incoming configuration response parsing phase of L2CAP. This allowed an attacker to utilize any arbitrary sized response to illegally overwrite data within the program. The vulnerable function *l2cap_parse_conf_rsp* is shown in Figure 3. The function receives a configuration response buffer (*rsp*) and its

[2]These line numbers were generated using David A. Wheeler's 'SLOCCount'.

Table 2. SEESAW Bluetooth target reaching times. The timeout used for all the execution was 6 hours (21600 seconds). "-" denotes target could not be reached within 6 hours.

Target No.	#Entry points	SEESAW time(s)	Baseline Time(s)
1	7	52.12	280.51
2	1	1.80	2.46
3	5	875.43	-
4	1	28.83	-
5	1	27.35	-
6	1	26.22	-
7	1	29.02	-
8	1	4.83	4.44
9	1	4.13	5.11
10	1	4.11	6.66
11	1	48.22	-

Table 3. SEESAW USB target reaching times. The timeout used for all the execution was 6 hours (21600 seconds). "-" denotes target could not be reached within 6 hours.

Target No.	#Entry points	SEESAW time(s)	Baseline Time(s)
1	5	967.14	-
2	5	1308.50	-
3	7	2361.88	-
4	1	1.80	-
5	7	2833.76	-
6	7	267.12	-
7	7	3509.71	-
8	1	793.62	-
9	2	1434.23	-
10	2	1490.49	-
11	2	1588.58	-
12	2	890.62	-

length (*length*) as argument. The elements of the buffer are extracted by *l2cap_get_conf_opt* and are written to the buffer pointed by the *data* variable. After every write, the pointer *ptr* is advanced, and it indicates the destination of the next element from the response buffer. The vulnerability arises from the lack of a check on the destination buffer size. Therefore, a crafted large response can cause out of bound memory access. SEESAW detected the vulnerability in the *l2cap_get_conf_opt* from the calling context in *l2cap_parse_conf_rsp* under the case *L2CAP_CONF_EFS* (line 36 in Figure 3). It took SEESAW 2296.39 seconds to detect the out of bound memory access while baseline symbolic execution could not reach the bug location within the 6 hours timeout.

Reducing False Positives. SEESAW reported four false positives within the Bluetooth module. These were manifested due to a lack of the calling context. These are mainly due to the static analysis component's reporting of all entry points that may lead to a particular protocol relevant target. However, some of these entry points may not provide a precise context. Such false positives can be reduced by manually restricting the entry points. For the USB module, SEESAW did not report any false positives.

4 Related Work

SVF [12] is a scalable static analysis tool with precise points-to analysis. However, it cannot detect memory overflow errors and cannot reason about pointer arithmetic that goes outside the boundary of an object to get its container. SEESAW leverages symbolic execution to detect memory overflow vulnerabilities and mitigates the non-standard pointer arithmetic problem by using API model guided symbolic execution to resolve the function pointer targets and passing them to the static analysis phase to explore additional paths. K-miner [8] builds on the LLVM compiler suite to analyze commodity operating system kernels like Linux. K-miner reduces the number of relevant paths by using system call interfaces as entry point, this partitions the kernel along

```
1  static int l2cap_parse_conf_rsp(struct l2cap_chan *chan,
   ↪  void *rsp,
2   int len, void *data, u16 *result) {
3   struct l2cap_conf_req *req = data;
4   void *ptr = req->data;
5   int type, olen;
6   unsigned long val;
7   struct l2cap_conf_rfc rfc = { .mode = L2CAP_MODE_BASIC
    ↪  };
8   struct l2cap_conf_efs efs;
9
10  BT_DBG("chan %p, rsp %p, len %d, req %p", chan, rsp,
    ↪  len, data);
11
12  while (len >= L2CAP_CONF_OPT_SIZE) {
13       len -= l2cap_get_conf_opt(&rsp, &type, &olen,
         ↪  &val);
14
15     switch (type) {
16       case L2CAP_CONF_MTU:
17         ...
18         l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2,
           ↪  chan->imtu);
19         break;
20
21       case L2CAP_CONF_FLUSH_TO:
22         chan->flush_to = val;
23         l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
24                            2, chan->flush_to);
25         break;
26         ...
27       case L2CAP_CONF_EFS:
28         if (olen == sizeof(efs))
29             memcpy(&efs, (void *)val, olen);
30
31         if (chan->local_stype != L2CAP_SERV_NOTRAFIC &&
32             efs.stype != L2CAP_SERV_NOTRAFIC &&
33             efs.stype != chan->local_stype)
34               return -ECONNREFUSED;
35
36         l2cap_add_conf_opt(&ptr, L2CAP_CONF_EFS,
           ↪  sizeof(efs),
37                            (unsigned long) &efs);
38         break;
39
40       case L2CAP_CONF_FCS:
41         ...
42       }
43   }
44     ...
45   return ptr - data;
46 }
```

Figure 3. Code snippet related to Bluetooth vulnerability

separate execution paths. An inclusion-based pointer analysis is done to resolve constraints of the function pointers. A flow sensitive pointer-analysis is done to improve precision further. Our work differs from K-miner by utilizing static analysis to gain the protocol relevant context. We are able to automatically identify user provided targets and relevant entry points are determined based on the protocol domain knowledge. Sys [4] combines light-weight static analysis with incomplete symbolic execution, which may skip parts of the code inside a function. SEESAW, on the other hand, uses precise static analysis to guide API model guided symbolic execution. While manual specification in Sys is at the static analysis stage, in SEESAW manually specified API models are used during symbolic execution to explore system-level code components in a precise and scalable way.

There have been prior tools that have combined symbolic execution with static analysis. Woodpecker [6] can speedup path exploration of symbolic execution by avoiding paths that are not related to a user provided checker. It cannot perform program modeling and must find a complete path starting from the main function. SEESAW is able to utilize any entry point in the program. Our configurations allow the targets and entry points to be decided by domain-model or it can also be user specified.

Dowser [9] combines guided fuzzing with program analysis and symbolic execution. It targets deep buffer-overflow vulnerabilities in program logic. It uses static analysis to identify targets. A combination of fuzzing and symbolic execution is used to steer execution towards the targets. Compared to Dowser our targets are not strictly typed. Our targets are found based on value-flow in the program and, therefore, SEESAW is able to explore the relevant input space without being restricted by the fuzzed inputs.

5 Conclusion

We presented SEESAW, a framework for detecting vulnerabilities in protocol stack implementations using API model guided extraction of protocol knowledge. It combines two major program analysis techniques: static analysis and symbolic execution. Both analysis techniques interact with each other and share the analysis results that benefit the whole analysis. Static analysis uses protocol information to find out relevant targets. Symbolic execution's scalability issues are handled by prioritizing paths that follow the target paths provided by static analysis. We utilize the precise memory model of symbolic execution in aiding static analysis for resolving indirect callsites. The components can interact iteratively to improve the precision of the analysis. Our methodology has been applied to the USB and Bluetooth protocol stack implementations in the Linux kernel. Our domain-knowledge guided analysis can detect memory vulnerabilities in protocol stack implementations in a scalable way and with up to a 99% speedup. Thus, SEESAW can be effectively used in the attack surface analysis of an IoT framework.

Acknowledgments

This work was partially funded by the US National Science Foundation under awards CNS-1815883 and CNS-1942235 and by the Semiconductor Research Corporation.

References

[1] [n. d.]. BlueBorne. https://www.armis.com/blueborne/.

[2] [n. d.]. CVE-2017-1000251 Detail. https://nvd.nist.gov/vuln/detail/CVE-2017-1000251.

[3] [n. d.]. Static Value-Flow Analysis Framework for Source Code. https://github.com/SVF-tools/SVF.

[4] Fraser Brown, Deian Stefan, and Dawson R. Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 199–216.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 209–224.

[6] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. *ACM SIGPLAN Notices* 48, 4 (2013), 329–342.

[7] Dawson Engler and Daniel Dunbar. 2007. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 1–4.

[8] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux.. In *NDSS*.

[9] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 22nd USENIX Security Symposium*. 49–64.

[10] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed symbolic execution. In *International Static Analysis Symposium*. Springer, 95–111.

[11] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 49–64.

[12] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[13] Tuba Yavuz and Ken Yihang Bai. 2020. Analyzing system software components using API model guided symbolic execution. *Autom. Softw. Eng.* 27, 3 (2020), 329–367.