Automated Detection of Spatial Memory Safety Violations for Constrained Devices

Sören Tempel[1] Vladimir Herdt[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
tempel@uni-bremen.de, vherdt@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—Software written for constrained devices, commonly used in the *Internet of Things* (IoT), is primarily written in C and thus subject to vulnerabilities caused by the lack of memory safety (e.g. buffer overflows). To prevent these vulnerabilities, we present a systematic approach for finding spatial memory safety violations in low-level code for constrained embedded devices. We propose implementing this approach using SystemC-based *Virtual Prototypes* (VPs) and illustrate an architecture for a non-intrusive integration into an existing VP. To the best of our knowledge, this approach is novel as it is the first for finding spatial memory safety violations which addresses challenges specific to constrained devices. Namely, limited computing resources and utilization of custom hardware peripherals. We evaluate our approach by applying it to the IoT operating system RIOT where we discovered seven previously unknown spatial memory safety violations in the network stack of the operating system.**

***Index Terms*—Symbolic Execution, Memory Safety, Embedded Software, Constrained Devices, Virtual Prototype, RISC-V, RIOT**

I. Introduction

Software for constrained devices, as used in the low-end *Internet of Things* (IoT), is primarily written in C [1, Table 1]. Bormann *et al.* define constrained devices as “small devices with limited CPU, memory, and power resources”. These constraints exist to make building large IoT networks consisting of numerous constrained devices financially viable. For example, while non-constrained conventional devices (e.g. laptops) have several gigabytes of memory at their disposal, constrained devices only have access to a few kilobytes of memory [2]. In the IoT context, these devices are often used in conjunction with custom hardware peripherals, such as sensors, to gather information about their surrounding environment.

The C programming language is a popular choice for programming these devices as it gives programmers control over low-level machine details thereby enabling optimization to reduce the use of scarce resources. Unfortunately, C is an inherently unsafe programming language, i.e. the C language specification leaves some behavior undefined. The classic example of undefined behavior in C is its lack of memory safety (e.g. buffer overflows or use-after-frees). Undefined behavior can be exploited by a malicious attacker to gain control over the constrained device [3]. For instance, stack-based buffer overflows can be utilized to subvert program control flow through an overwrite of the function return address as stored in the stack frame.

For conventional devices a variety of tools are available for detecting memory safety violations and other sources of undefined behavior during early stages of software development. Most C programmers will be familiar with development tools such as Valgrind [4] or AddressSanitizer [5] which are often used for this exact purpose on non-constrained devices. Unfortunately, neither Valgrind nor AddressSanitizer are intended to be used on bare-metal and instead target conventional operating systems such as Linux.

Prior work by Devietti *et al.* has proposed HardBound [6], a technique for detecting memory safety violations in C code which we believe to be applicable to constrained devices. HardBound performs runtime boundary checks in hardware using a custom hardware peripheral [6]. Unfortunately, such custom hardware is difficult to manufacture and thus not commonly available early on in the development process.

In order to employ HardBound as an early software testing technique, we propose combining it with *Virtual Prototypes* (VPs). VPs provide an executable model of the entire hardware platform and are commonly created in SystemC[1] TLM (*Transaction-Level Modeling*) [7]. As such, VPs enable early simulation and testing of software targeting this platform. To ensure comprehensive test coverage during VP-based software testing, we combine HardBound with symbolic execution, a software testing technique which enumerates reachable program paths based on symbolic input variables. This combination allows us to check all reachable programs paths (obtained through symbolic execution) for spatial memory safety violations (via application of Hardbound on each path) in a VP-based execution environment.

Software for constrained devices interacts closely with hardware peripherals and often relies on custom peripherals. By combining HardBound with VPs, we can model these peripherals using SystemC TLM and check low-level code interacting with them for spatial memory safety violations. To the best of our knowledge, this is a novel feature of our approach which is not supported by prior approaches (section III). We demonstrate that an integration of HardBound with VPs and symbolic execution is feasible (section IV). Afterwards, we evaluate our integration by conducting experiments with RIOT, a popular operating system designed explicitly for constrained devices (section V). We tested the network stack of the RIOT operating system, including low-level network driver code, using our HardBound extended VP. As part of these experiments, we found seven previously unknown memory safety violations in the RIOT network stack. They have been acknowledged by RIOT developers and are currently in the process of being fixed. To stimulate further research we have released our HardBound extended VP on GitHub[2].

[1]SystemC is a C++ class library for modeling hardware.
[2]https://github.com/agra-uni-bremen/hardbound-vp

```
1  load_addr R2, 0x1000
2  setbound R2, 0x1000, 4
3
4  load_byte R3, R2 // load at 0x1000, success
5  addi R2, R2, 2   // R2: (0x1002, 0x1000, 0x1004)
6  load_byte R3, R2 // load at 0x1002, success
7  addi R2, R2, 2   // R2: (0x1004, 0x1000, 0x1004)
8  load_byte R3, R2 // load at 0x1004, fail
```

Fig. 1. HardBound example usage [6, Figure 2].

II. Background

The following subsections serve as a brief primer on memory safety, HardBound, and symbolic execution as a software testing technique.

A. Memory Safety

Existing publications on memory safety issues of the C programming language distinguish spatial memory safety and temporal memory safety as follows [8, p. 53]:

> *Temporal safety* is ensured when memory is never used after it is freed. *Spatial safety* is ensured when any pointer dereference is always within the memory allocated to that pointer.

The programming language C offers neither spatial nor temporal memory safety. Lack of memory safety allows attackers to perform unintended computations which are central to many security vulnerabilities. Prior work by Szekeres *et al.* provides a formal model for attacks exploiting memory safety issues [3]. A variety of different techniques have been proposed by prior work which attempt to address the lack of memory safety in the C programming language [6], [8], [9]. As explained in the Introduction, our approach is based on a combination of HardBound and symbolic execution, both will be further described in the following subsections.

B. HardBound

HardBound enforces spatial memory safety for C programs through a hardware peripheral. Enforcement is achieved by enhancing values representing C pointers with bounds information tracked in hardware. Contrary to software-only approaches, HardBound does not modify the C pointer representation and allows bounds checks to be performed efficiently in hardware. Conceptually, each register and memory value in HardBound is a triplet $(value, base, bound)$ where $value$ represents the original pointer value and $base/bound$ represent the lower/upper bound of the bounded pointer. We will refer to the additional $base$ and $bound$ information as HardBound metadata in the following. The HardBound metadata is used by a custom hardware peripheral to perform boundary checks on each load/store instruction. As such, it is ensured that each load/store is within the pointer bounds as specified by the $base$ and $bound$ metadata. For this reason, spatial violations cannot occur on bounded pointers [6].

Since the C type system is not accessible at the binary level, the executed software must communicate which values represent pointers (and their respective HardBound metadata) to the hardware peripheral. In the original HardBound paper this is achieved through a custom `setbound` instruction. The insertion of `setbound` instructions is automated using "simple intra-procedural compiler instrumentation" [6, p. 103]. The hardware peripheral is in turn responsible for propagating metadata initial set by the executed software. As an example, consider pointer arithmetic as performed using an `addi` instruction in the pseudo assembler code in Figure 1. In this example, a pointer to a four byte value at address `0x1000` is created (Line 1 - 2). The pointer is then incremented (Line 5 and 7) and dereferenced (Line 4 and 6), on each increment the metadata must be propagated. Ultimately, the load in Line 8 fails as the pointer value is no longer within the propagated bounds.

C. Symbolic Execution

Symbolic execution is an automated software testing technique. Contrary to normal program execution, symbolic execution does not use concrete input values. Instead, input values are declared as symbolic variables and operations on these inputs cause the symbolic variables to be constrained accordingly. After execution terminates, a constraint solver is used to find new assignments for the inputs values based on the tracked constraints. The new input values, generated by the constraint solver, cause the execution of a different path through the program. For example, new input values might cause execution of a different if statement branch in the program. This process is repeated until all paths through the program have been explored, some predefined coverage goal has been reached, or an error is detected.

During execution of each discovered path, a path analyzer is used to determine whether the currently executed path constitutes an error. In this publication, we provide a path analyzer for finding spatial memory safety violations. The analyzer we are proposing is specifically tailored to constrained devices and addresses challenges specific to these devices which have not been addressed in prior work. We use this analyzer in conjunction with an open source symbolic execution engine from prior work done by Tempel *et al.* (referred to as `symex-vp` in the following) [10].

III. Related Work

Prior work has largely focused on the detection of spatial memory safety violations on conventional devices. Popular approaches in this regard include EXE [11], which utilizes compiler instrumentations to detect spatial memory safety violations, and KLEE [12] which symbolically executes LLVM IR, the intermediate language used by the LLVM compiler infrastructure. As such, KLEE-based approaches do not capture low-level machine details and operate on a higher abstraction level than our approach which symbolically executes machine code directly.

With the focus on conventional systems, the aforementioned publications also do not address challenges specific to constrained devices as discussed in the Introduction. Prior work by Muench *et al.* discusses these challenges further [13]. They identified "silent memory corruptions" as the predominant issue in this domain, most of which would be detected by conventional operating systems through employed protection mechanisms lacking on constrained devices to reduce production costs [13, p. 13]. They also propose several heuristics to improve error detection in this domain and combine these heuristics with fuzzing to automatically discover memory corruptions [13, p. 9]. Our approach does not rely on heuristics and we believe symbolic execution to be preferable over fuzzing on constrained devices as the state space is smaller due to limitations on code size. This might mitigate the state explosion problem known from conventional devices [14, p. 4].

This hypothesis is confirmed by Davidson *et al.* who present FIE, a symbolic execution framework for finding vulnerabilities in embedded firmware [15]. FIE targets MSP430 microcontrollers and attempts to achieve a "complete analyses for simple firmware programs" [15, p. 467]. FIE is based on KLEE and therefore also executes LLVM IR symbolically. For this reason, FIE operates on a higher abstraction level than our own approach which symbolically executes RISC-V machine code directly. This comes with the drawback that FIE cannot take low-level machine details into consideration. As an example, FIE fails to execute paths which include inline assembly, usage of which we believe to be common on constrained devices. Furthermore, FIE does not use accurate models of hardware peripherals. Instead, it approximates peripheral behavior through given memory and interrupt specifications which results in potential false-positives [15, p. 476]. Since our approach is based on SystemC, we have existing models of peripherals at our disposal and can easily model new ones.

Lastly, prior work by Herdt *et al.* provides symbolic execution of embedded binaries [16]. However, this publication does not support unmodified SystemC peripherals. Furthermore, the employed path analyzer is only capable of detecting spatial violations in buffers allocated dynamically through `malloc`. Contrary to our own approach, this publication therefore misses overflows in data structure not allocated dynamically (e.g. buffers allocated on the stack) [16, Section 4.4.2].

We believe that our novel VP-based approach for detecting spatial memory safety violations in software designed to be used on constrained devices mitigates the discussed pitfalls we identified in prior work.

IV. VP-based HardBound Integration

In the following subsections we will describe how HardBound support can be integrated into a standard SystemC TLM-2.0 architecture. As per subsection II-C, we extended the open source SystemC-based `symex-vp`[3]. This VP targets the RISC-V architecture and allows symbolic execution of `RV32IMC` machine code.

A. Overview

Figure 2 provides an overview of our proposed architecture. As explained in subsection II-C, both the *Instruction Set Simulator* (ISS) and the symbolic execution engine are provided by `symex-vp`. The latter provides us with symbolic types, a path explorer for finding new paths through the executed software, and an SMT Solver for solving constraints on symbolic types. In order to employ HardBound as a path analyzer, we had to integrate it with the ISS. By relying on implicit C++ type conversion we were able to keep required modifications minimal, ultimately only extending around 700 LOC in `symex-vp`. This illustrates that a non-intrusive integration is possible.

The ISS is the main component of the VP, it is responsible for fetching, decoding, and executing RISC-V instructions.

[3]https://github.com/agra-uni-bremen/symex-vp

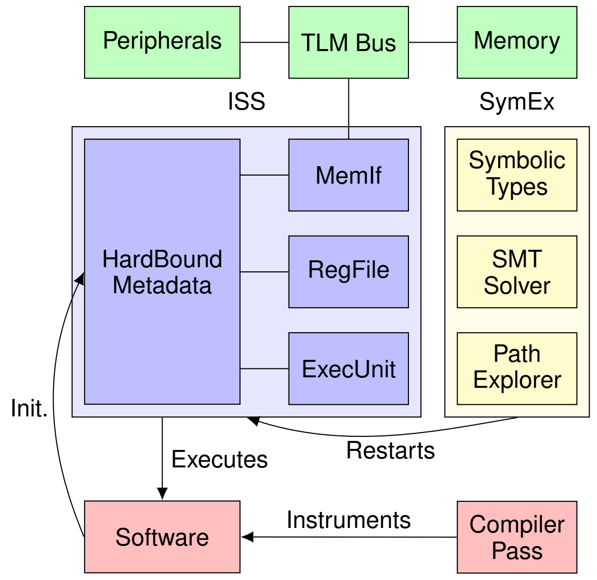

Fig. 2. Overview of our HardBound implementation for `symex-vp`.

As such, it executes a given RISC-V software provided in binary form. Execution is based on symbolic input values, supplied by the symbolic execution engine (referred to as SymEx in Figure 2). After software termination, the symbolic execution engine determines new assignments for symbolic input variables and restarts the ISS—and thereby also the software—with these new variable assignments. New variable assignments result in the discovery of new paths through the executed software. On each path, performed load/store instructions are bounds checked using the provided HardBound metadata (see subsection II-B). This metadata is initialized by the executed software through special instructions inserted by a compiler pass. The ISS is responsible for propagating it correctly during execution as illustrated in subsection II-B.

B. Metadata Propagation

Since our work is based on `symex-vp`, our ISS supports symbolic execution. As such, instruction operands may represent symbolic values. We modified the ISS to also track and propagate HardBound metadata alongside these symbolic values. During software execution, the ISS interacts with the stored metadata to perform bounds checks. Recall that HardBound metadata is only required for values representing C pointers. These values may be stored in either memory or registers. As such, the register file (RegFile) and the memory interface (MemIf) had to be modified (see center of Figure 2). The register file is responsible for storing register values. The memory interface is responsible for interactions with memory-mapped peripherals through the SystemC TLM bus. We modified both—the memory interface and the register file—to ensure they store associated HardBound metadata for memory and register values representing pointers. The main execution unit (ExecUnit) of the ISS accesses this metadata and propagates it when executing instructions which are used to implement C pointer arithmetic. For example, an instruction adding a constant to a bounded pointer value must itself return a bounded pointer value with associated HardBound metadata (recall the example from Figure 1).

Implementation of metadata propagation was the most central change made to symex-vp as we had to switch the underlying data type, used by the execution unit for instruction operands, from symbolic expressions to a tuple which additionally tracks the HardBound metadata. The modifications required for this change were kept to a minimum by relying on implicit C++ type conversions, thereby allowing implicit conversions from symbolic expressions—with associated HardBound metadata—to plain symbolic expressions. As the majority of RISC-V instructions are not commonly used to manipulate pointer values, we were able to refrain from modifying the implementation of these instructions. As such, only the implementation of instructions which are used by C compilers to implement pointer arithmetic had to be modified in the execution unit. In the RISC-V RV32IMC context we thus implemented metadata propagation for the following instructions: ADD, ADDI, and SUB. In the following subsection we will explain how propagated metadata interacts with SystemC TLM in the HardBound context.

C. TLM Integration

Storing and propagating HardBound metadata in the ISS allows us to perform bound checks on load/store instructions. In the SystemC context, these instructions are implemented through SystemC TLM based on a bus abstraction. The ISS communicates with devices attached to the TLM bus (e.g. memory or memory-mapped peripherals) using TLM transactions created by a memory interface [7, p. 430]. In order to avoid modifications of peripherals attached to the TLM bus, we are not propagating HardBound metadata over TLM and instead perform a transparent conversion within the memory interface itself using an internal mapping $\delta\colon \mathit{addr} \mapsto \{(\mathit{base}, \mathit{bound})\}$. HardBound metadata (*base* and *bound*) can be updated by storing new metadata at *addr* using a store instruction. Load instructions return HardBound metadata if a mapping $\mathit{addr} \to (\mathit{base}, \mathit{bound})$ exists in δ for the loaded *addr*. HardBound metadata originates in the executed software through explicit setbound instructions.

While the ISS is responsible for storage and propagation of HardBound metadata it is incapable of initializing the metadata as information regarding pointer bounds is difficult to infer at the binary level. Instead, the software itself communicates initial metadata values to the ISS upon pointer creation. For each created pointer, the instructions required for communicating associated bounds metadata to the ISS are automatically inserted into the tested software by a compiler pass.

D. Compiler Pass

A compiler pass is used to automatically initialize HardBound metadata for pointers to local or global variables. The compiler pass performs an analysis detecting the creation of pointers, infers the size of the values pointed to, and communicates this information to the VP.

In order to ease supporting memory-mapped I/O, we deviated from the compiler pass implementation in the original HardBound paper regarding the handling of pointer casts. Normally, creating a pointer from an integer (e.g. (int *)0x1000) is an unsafe operation as no bounds information is associated with the memory address 0x1000 [6, p. 112].

```
1  static char buf[BUFFER_SIZE];
2
3  int add_to_buffer(char c) {
4    static size_t index = 0;
5    if (index >= BUFFER_SIZE)
6      return -1;
7
8    // ------- [[ Original Code ]] -------
9    buf[index] = c;
10   // ------- [[ 1st Compiler Pass ]] -------
11   char *ptr = &buf[0];
12   *(ptr + index) = c;
13   // ------- [[ 2nd Compiler Pass ]] -------
14   char *ptr = &buf[0];
15   setbound(&ptr, ptr, sizeof(buf));
16   *(ptr + index) = c;
17   // ------- END -------
18
19   index++;
20   return 0;
21 }
```

Fig. 3. Performed HardBound compiler pass transformations.

This is, however, a common idiom to communicate with memory-mapped peripherals from low-level C code. For this reason, we relaxed the handling of these unsafe casts in the compiler pass. This prevented the addition of manual setbound invocations for memory-mapped peripheral accesses.

Furthermore, the original HardBound paper does not provide a detailed description of how direct array accesses are handled in the compiler pass. Consider an array access such as buf[i] = n, since no pointer exists in this example code no HardBound metadata is available for ensuring spatial memory safety. To mitigate this problem, we have written two compiler passes. The first transforms any array access of the form buf[i] = n to a pointer-based access of the form *(buf + i) = n. The second pass communicates pointer bounds, upon pointer creation, to the VP.

Figure 3 illustrates the transformations performed by the two compiler passes. The original code (Line 9) performs a direct array access on buf. The first compiler pass rewrites this to a pointer-based access (Line 11 - Line 12). The second compiler pass inserts an appropriate call to a setbound function which communicates the bounds of ptr to the VP[4] (Line 14 - Line 16). We have implemented both passes using LLVM and published them as open source software on GitHub[5].

V. Evaluation

We evaluate our approach by applying it to the RIOT operating system. RIOT is targeting constrained devices as defined in RFC 7228 [2] (i.e. the low-end Internet of Things). We used RIOT as an evaluation target since prior work by Hahm *et al.* considers it to be one of the “most prominent open source” operating systems in this domain [1, p. 732]. Furthermore, RIOT employs a code quality management process and automated unit tests, thereby aiming for high code quality [17, p. 4438]. This allows us to evaluate whether our approach is capable of finding real bugs missed during

[4]On a technical note, we intercept RISC-V ecall instructions in the VP to set bounds information from the software. The original HardBound paper uses custom instructions.

[5]https://github.com/agra-uni-bremen/hardbound-llvm

TABLE I
SPATIAL MEMORY SAFETY VIOLATIONS FOUND IN RIOT MODULES.

Id	Module	Test	#Paths	Time	#instr
#15927	`uri_parser`	UNIT	48	11 s	408646
#15930	`uri_parser`	UNIT	156	35 s	1311034
#15945	`clif`	UNIT	227	50 s	1847765
#15947	`clif`	UNIT	10	2 s	91302
#16018	`gnrc_rpl`	SLIP	75	143 s	3378636
#16062	`gnrc_rpl`	SLIP	72	307 s	3444769
#16085	`gnrc_rpl`	SLIP	855	3532 s	46262378

manual code review and unit testing. RIOT supports a variety of hardware platforms. For our experiments we utilize the constrained SiFive HiFive1[6] platform which uses RISC-V and is supported by both RIOT and `symex-vp`. RIOT itself is further described in a publication by Baccelli *et al.* [17].

In accordance with prior work, we believe input handling routines of the network stack to be the biggest attack vector of a networked IoT operating system [18]. Our experiments therefore focus on RIOT components which are part of this network stack. In the following we will further describe how we employed HardBound in the RIOT context to analyze these components and which memory safety violations we were able to uncover through our analysis.

A. RIOT HardBound Setup

HardBound is intended to be deployable with "minimally invasive changes to the compiler and run-time". As a first step, we had to ensure that the RIOT build system utilizes our LLVM-based compiler pass. Fortunately, RIOT already supports compilation with LLVM. For this reason, we only had to modify the employed compiler flags via a build system configuration variable. The original HardBound paper also acknowledges that library functions performing memory allocations, e.g. `malloc`, need to be modified to include appropriate `setbound` calls [6, p. 107]. While usage of `malloc` in RIOT is discouraged, we still had to modify the RIOT module used to allocate memory for network packets to set the appropriate bounds for each returned packet. This allows us to discover spatial violations potentially occurring when accessing these packets.

Overall, our HardBound setup for the RIOT operating system was straightforward and only required the outlined changes which we believe to be negligible in terms of effort required. For our experiments we boot RIOT on our HardBound extended VP and perform boundary checks in the VP during the execution of RIOT software. More details are provided in the next subsection.

B. Results

RIOT follows a modular software architecture, modules which should be enabled are selected at compile-time [17, p. 4430]. We tested several RIOT modules which are part of the network stack using our proposed software testing technique. In this regard, we distinguish two test types:

1) UNIT tests, conducted using custom test drivers which invoke functions from the public module API. In this case, symbolic values are created directly in the test driver through custom instructions[7].
2) SLIP tests, conducted using existing RIOT example applications. Symbolic values are introduced through a custom SLIP [19] network interface which is implemented in the VP.

We tested different modules of the RIOT network stack using both approaches. We used UNIT tests for utility modules, used indirectly by network protocol implementations. SLIP tests were used for freestanding implementations of network protocols, which directly process input received through the *Internet Protocol* (IP).

We tested the following RIOT network modules which implement internet protocols that we believe to be in common use in the low-end IoT context on constrained devices:

1) The `uri_parser` module which provides a non-destructive parser for *Uniform Resource Identifier* (URI) references as defined in RFC 3986 [20].
2) The `clif` module which provides a parser for the CoRe Link Format as used in REST architectures for constrained devices and defined in RFC 6690 [21].
3) The `gnrc_rpl` module which provides an implementation of the IPv6 *Routing Protocol for Low-Power and Lossy Networks* (RPL) as defined in RFC 6550 [22].

In total, we found seven previously unknown spatial memory safety violations in the tested modules, all of which have been discovered using our proposed combination of symbolic execution and HardBound. We have reported these issues to RIOT developers and they are currently in the process of being fixed. Further information regarding the discovery of individual issues is provided in Table I. For each discovered issue, we list the identifier in the public RIOT issue tracker[8], the module in which it was found, the employed test method, the number of paths enumerated until it was found, and the total execution time. As a complexity metric, we also include the total amount of RISC-V instruction executed.

In modules which we tested through UNIT tests, spatial memory safety violations are discovered faster and fewer instructions are executed. This is due to the fact, that less constrains are tracked as we only test individual functions. With SLIP tests, issue discovery takes longer as the input is passed through the entire network stack, thereby imposing more constrains on symbolic input variables. However, given the complexity of a routing protocol (such as RPL) this indicates that our approach is also capable of finding spatial memory safety violations in complex real-world code for constrained devices.

VI. CONCLUSION

In this paper, we concerned ourselves with the early detection of spatial memory safety violations on constrained devices. We proposed and implemented a VP-based software testing technique for this purpose which uses symbolic execution to enumerate reachable program paths and checks each path for spatial violations through a SystemC-based HardBound implementation (section IV). We illustrated an architecture for achieving a non-intrusive integration of HardBound with existing SystemC-based VPs and implemented this architecture on top of the open source `symex-vp`. We have also released our HardBound extended version of

[6]https://www.sifive.com/boards/hifive1

[7]We intercept RISC-V `ecall` instructions in the VP for this purpose.

[8]https://github.com/RIOT-OS/RIOT/issues

TABLE II
EXTENDED DESCRIPTION OF INDIVIDUAL SPATIAL MEMORY SAFETY VIOLATIONS FOUND IN RIOT NETWORK MODULES.

Module	Bug Description
`uri_parser`	**#15927:** During parsing of the *userinfo* part of a URI the parser did not check if the input is long enough to even contain a complete *userinfo* if the URI contained an `@` character, thereby performing an out-of-bounds read on the provided input buffer. **#15930:** The `uri_parser` module attempted to parse data after the *hier-part* of the URI, even if none was present. For example, on an input like `a://` the parser would perform an out-of-bounds read due to missing bounds checks.
`clif`	**#15945:** During parsing of key-value pairs, the module did not check if a value was actually present after the key has been read, thereby performing an out-of-bounds read. **#15947:** During parsing of link attributes, the `clif` module did not check whether any attributes were present, thus performing an out-of-bounds read on the input buffer.
`gnrc_rpl`	**#16018:** RPL messages are parsed by casting buffers to packed structs. Unfortunately, `gnrc_rpl` did not check if the buffer was large enough to contain the struct in some instances. **#16062:** During validation of RPL options, `gnrc_rpl` did not check if the input is large enough to contain a given option. Attempts to access this option resulted in an out-of-bounds read. **#16085:** The `gnrc_rpl` module separates option parsing from option validation. Similar to #16062, the option parsing code was also lacking proper bounds checks, resulting in an out-of-bounds read.

`symex-vp` on GitHub to stimulate further research. Our main contribution is a new path analyzer for an existing symbolic execution engine. As per section III, our approach is novel as it addresses challenges specific to constrained devices not addressed in prior work. We applied our approach to an operating system for constrained devices (RIOT), were we found seven previously unknown bugs, which have been missed by unit testing and manual code review (section V).

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001.

REFERENCES

[1] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, Oct. 2016, ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2505901.

[2] C. Bormann, M. Ersue, and A. Keränen, *Terminology for Constrained-Node Networks*, RFC 7228, May 2014. DOI: 10.17487/RFC7228.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[4] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100, ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA: USENIX, 2012, pp. 309–318.

[6] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural Support for Spatial Safety of the C Programming Language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 103–114, ISBN: 9781595939586. DOI: 10.1145/1346281.1346295.

[7] System C Standardization Working Group, "IEEE Standard for Standard SystemC Language Reference Manual," IEEE, Tech. Rep., 2012, pp. 1–638. DOI: 10.1109/IEEESTD.2012.6134619.

[8] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C Safe by Extension," in *2018 IEEE Cybersecurity Development (SecDev)*, Sep. 2018, pp. 53–60. DOI: 10.1109/SecDev.2018.00015.

[9] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," *SIGPLAN Not.*, vol. 44, no. 6, pp. 245–258, Jun. 2009, ISSN: 0362-1340. DOI: 10.1145/1543135.1542504.

[10] S. Tempel, V. Herdt, and R. Drechsler, "An Effective Methodology for Integrating Concolic Testing with SystemC-based Virtual Prototypes," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, Feb. 2021, pp. 218–221. DOI: 10.23919/DATE51398.2021.9474149.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008, ISSN: 1094-9224. DOI: 10.1145/1455518.1455522.

[12] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[13] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *The Network and Distributed System Security Symposium 2018*, ser. NDSS, 2018.

[14] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657.

[15] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX Association, Aug. 2013, pp. 463–478, ISBN: 978-1-931971-03-4.

[16] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, Las Vegas, NV, USA: Association for Computing Machinery, 2019, ISBN: 9781450367257. DOI: 10.1145/3316781.3317807.

[17] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018, ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2815038.

[18] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The Halting Problems of Network Stack Insecurity," *USENIX ;login:*, vol. 36, no. 06, pp. 22–32, 2011.

[19] J. Romkey, *Nonstandard for transmission of IP datagrams over serial lines: SLIP*, RFC 1055, Jun. 1988. DOI: 10.17487/RFC1055.

[20] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, RFC 3986, Jan. 2005. DOI: 10.17487/RFC3986.

[21] Z. Shelby, *Constrained RESTful Environments (CoRE) Link Format*, RFC 6690, Aug. 2012. DOI: 10.17487/RFC6690.

[22] R. Alexander, A. Brandt, J. Vasseur, J. Hui, K. Pister, P. Thubert, P. Levis, R. Struik, R. Kelsey, and T. Winter, *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*, RFC 6550, Mar. 2012. DOI: 10.17487/RFC6550.