Fuzzing Embedded Systems: An Investigation Into Custom Memory Allocators and Automated Software Testing

Spencer Hua
Vertically Integrated Projects
Georgia Institute of Technology
spencerhua@gatech.edu

Ammar Ratnani
Vertically Integrated Projects
Georgia Institute of Technology
aratnani7@gatech.edu

Zelda Lipschutz
Vertically Integrated Projects
Georgia Institute of Technology
hlipschutz3@gatech.edu

Chris Reid
Vertically Integrated Projects
Georgia Institute of Technology
creid61@gatech.edu

Allen Stewart
Vertically Integrated Projects
Georgia Institute of Technology
allen.stewart@gtri.gatech.edu

***Abstract*—With the increasing size and complexity of modern software, manually auditing source code for vulnerabilities has become intractable for all but the smallest programs. As a result, automated software techniques like fuzzing have become very popular due to their effectiveness at finding unique crash paths. This paper is a summary of the authors' attempts at applying modern fuzzing methods to embedded systems. Specifically, it looks at fuzzing `malloc` implementations, including those provided by musl, uClibc, and AVR Libc. Additionally, the paper examines the authors' progress in identifying the fundamental bugs found by fuzzing, as well as their efforts to create a functional remote code execution (RCE) exploit.**

***Index Terms*—Fuzzing, Cybersecurity, Embedded Systems, Heap Exploitation**

I. Introduction

In present times, the ubiquity of embedded devices loaded with programs that require dynamic memory allocation makes rigorous testing critical to eliminate as many vulnerabilities as possible. In recent decades, fuzzing has established itself as a powerful software vulnerability detection tool. Fuzzing refers to automatic test generation which pass semi-random, malformed inputs to programs with the goal of causing unexpected behavior that can be exploited. It is especially effective in detecting vulnerabiities that can go undetected through static program analysis or manual code inspection methods, like penetration testing [1].

Fuzzing is currently used to detect bugs in many popularly used programs. Google's fuzzing tool ClusterFuzz, which was open-sourced in 2019, found over 16,000 bugs in Chrome and over 11,000 bugs in open source projects [2]. Microsoft has also incorporated fuzzing into its Security Development Lifecycle, requiring "fuzzing at every untrusted interface of every product" [1]. Even though there has been extensive testing through fuzzing of commonly used programs and libraries like glibc on popular architectures like x86, there is still a dearth of testing for alternative implementations on relative obscure RISC architectures like AVR, which are primarily used in embedded systems. We attempt to bridge this gap by fuzzing the dynamic memory allocation function `malloc` on an AVR microcontroller.

II. History and Benefits

The term "fuzz" was first coined in 1988 when Professor Barton Miller remotely logged into a Unix system during a storm, which caused so much interference on the dial-up link that applications using the data off the line crashed [3]. Professor Miller then assigned a project to his class titled "Operating System Utility Program Reliability – The Fuzz Generator", where students were expected to create a basic command-line program to test the reliability of Unix programs by throwing random data at them and monitoring for any crashes. The earliest known fuzzer, "The Monkey", written by Steve Capps in 1983, created random mouse clicks and keyboard input to test MacWrite and MacPaint, which looked like an invisible monkey was using the computer.

Fuzzing has grown into a powerfully reliable testing technique with companies including Apple, Microsoft, Adobe, and Google publishing their fuzzing efforts [3]. Currently, the most popular fuzzer is American Fuzzy Lop, which uses compile-time means and genetic algorithms to crash programs and locate crashes for debugging purposes [4]. Some examples of significant bugs revealed by fuzzing include a possible Denial of Service attack on the Apple wireless internet drivers, known as MOKB-30-11-2006 [5].

The primary benefits of fuzzing come from its ability to increase code coverage to levels a human cannot match. For example, YouTuber LiveOverflow published a series on how the vulnerability CVE-2021-3156 in the ever-present `sudo` program could have been discovered via fuzzing, and how to subsequently develop an exploit [6]. Astonishingly, when the vulnerability was first discovered by Qualys, the underlying

bug had been present for over a decade, highlighting the necessity of better automated testing [7].

Finally, since fuzzing is such a versatile debugging tool, one possible application of fuzzing is in the annual CSAW Embedded Security Competition, as the competition is a main focus of the author's research program.

III. Research Targets

Since fuzzing is a broad category of fields, the authors are focusing on two specific categories of devices to apply their efforts.

A. Embedded Systems

Currently, the majority of fuzzing research has been focused on fuzzing programs and components designed for general-purpose operating systems, like the Linux kernel or Microsoft Windows. However, not as much research has been done into embedded systems, which can vary drastically from "normal" environments in their lack of a memory management unit (MMU), undocumented source code, or even custom closed-source operating systems. As Muensch et al. describe, many embedded systems without MMUs or with custom operating systems don't exhibit normal failure responses, instead silently malfunctioning or not exhibiting any error conditions at all [8]. This presents a difficult conundrum – how can a crash be debugged, fixed, or even recognized when the failure conditions are extremely opaque?

One potential solution is to write a high-quality emulator, as Pucher, Kudera, and Merzdovnik demonstrate in AVRS [9]. AVRS is a high-quality AVR emulator with a specific focus on reverse-engineering and a fuzzer built into the emulator. Due to the accuracy and speed of the emulator, AVRS can detect vulnerabilities that can present themselves in actual embedded AVR devices, like misused format string specifiers and return address stack smashing using a shadow call stack.

Alternatively, fuzzing efforts can make use of hardware abstraction layers (HALs), which are typically used during embedded development. Clements et al. published HALucinator, a firmware re-hoster that uses existing HALs to produce an extremely accurate emulator of many embedded systems [10]. To further demonstrate the accuracy, Clements et al. demonstrate the practicality by attaching the ever-popular AFL fuzzer to demonstrate that bugs can be accurately detected.

B. Dynamic Memory Allocation

Any sufficiently large application requires dynamic memory allocation. The flexibility it offers over static or stack-based allocation is indispensable, as is its performance with respect to both minimizing unnecessary copies and increasing memory utilization. The heap is pervasive (rightfully so), but that very pervasiveness combined with the difficulty of writing a good memory allocator unfortunately means heap exploits are rampant and devastating.

Perhaps the most famous one in recent memory involves `sudo`: CVE-2021-3156. Specifically, a buffer overflow on the heap can corrupt the metadata used by `malloc` to manage its free-lists. Worse still, an attacker can do so with arbitrary data of arbitrary length [11]. With that precondition, gaining access to a shell is relatively straightforward [12]. The case of `sudo` was particularly disturbing given the binary runs with root privileges, rendering a shell equivalent to unrestricted arbitrary code execution.

CVE-2021-3156 was most visible in the context of desktop and server systems, but that's not to say embedded applications don't face memory allocation bugs. In fact, one such exploit was discovered in late 2019 for The Legend of Zelda: Ocarina of Time for the Nintendo 64. Specifically, it's possible to `free` an object the player is holding. The player then tries to update the object via the pointer it retains, causing a use-after-free bug. By manipulating the allocation order, this method can modify the data of many game objects. In its simplest form, the exploit can be used to change the contents of chests. More excitingly, it can change objects' function pointers, allowing for a very restricted form of arbitrary code execution. It was enough to warp to the credits in Ocarina of Time, and warp to the final boss in The Legend of Zelda: Majora's Mask, which runs on the same engine [13], [14]. While this isn't a critical vulnerability, it demonstrates the pervasiveness of heap exploits. Even these games, which ran on the bare metal of the Nintendo 64, still used a heap to manage their memory, and heap bugs were still an entrypoint for attackers.

Heap issues on embedded systems are common enough to have been incorporated into many CTFs. One challenge involves exploiting a memory race condition. Some lock-free code was written assuming total store ordering between threads. However, ARM's memory model is much more lenient than x86, causing a race condition in the code whenever total store ordering is violated. This escalated into a use after free, which escalated into arbitrary reads and writes, and eventually a shell [15]. Another challenge proceeds in much the same way, using a double free to escalate to a shell. Unlike most challenges, it uses musl instead of glibc [16].

Although the primitives used by these exploits like double frees and use-after-frees aren't exclusive to embedded systems, the methods for obtaining then using them certainly are. Different processors, different C libraries, and sometimes the lack of supervision by an operating system equate to a unique landscape for heap bugs on embedded devices. To explore it, the authors chose dynamic memory allocation as a target for their research.

IV. Partial Results

In exploring memory allocation on embedded systems, the authors investigated a breadth of approaches. Unfortunately, not all of them gave results. Still, unsuccessful paths are worth documenting, and they are in this section.

A. The C Library

AFL is currently capable of fuzzing user-space binaries, but not much research has been done in connecting it to lower-level components [17]. In particular, `afl-gcc` is unable to successfully instrument and fuzz a system's C library. This

presented a problem for our research, as the `malloc` is usually tightly integrated with its libc. To remedy this, the authors put some effort into decoupling AFL from the underlying standard library.

The target C library chosen was musl, a popular alternative to glibc commonly used on embedded Linux systems, as well as being the default for Alpine Linux. This alternative was chosen due to musl's relative simplicity compared to glibc, as well as the ease of creating a toolchain based on it. Not only is it statically linkable, it also provides a script (`musl-gcc`) to use the system's C compiler to link with the musl standard library. This way, one doesn't have to rebuild the toolchain as one has to with uClibc. Additionally, AFL's least featureful GCC/CLANG mode was chosen to instrument the code. They're the simplest in operation, simply editing the generated assembly code before it's passed to the assembler. This simplicity makes it easy to make changes to the assembly payload if needed.

The primary difficulty in interfacing AFL with the C library is that it relies on system calls to setup fuzzing [18]. Specifically it relies on:

- `atoi`,
- `getenv`,
- `mmap`,
- `shm_open`,
- `shmat`,
- `write`,
- `read`,
- `close`,
- `fork`,
- `waitpid`, and
- `_exit`.

If any one of these functions has AFL instrumentation, then the setup routine `__afl_setup_first` will recurse infinitely. It will call one of these standard library functions, which will then call `__afl_maybe_log`, which will then call the setup routine again.

To avoid this outcome, either these particular functions have to be left uninstrumented or AFL has to be modified to guard against infinite recursion. The latter approach is more general and seems more tractable. One could hypothetically create a new `COMMON` variable to flag that `__afl_setup_first` is currently executing. The function would check that flag on entry and jump to `__afl_return` if it's set. Unfortunately, the authors decided to focus their efforts on alternative targets.

B. malloc-ng

As well, while `hangover` works with musl's `mallocng` implementation, there were no segmentation faults discovered, only abort signals [19]. The authors suspect that this may be due to faults on `hangover`'s end, rather than actual errors within `mallocng`.

C. AVR

As mentioned above, a high-quality emulator called AVRS was developed by Pucher, Kudera, and Merzdovnik. The authors were interested in accessing the source code of AVRS and reached out to Pucher, Kudera, and Merzdovnik, but the source release is still pending due to difficulties on the AVRS authors' end.

V. Results

At Georgia Tech, Homework 10 of CS 2110 taught by Caleb Southern and Dan Forsyth is to write a simple memory allocator. All of the authors had taken that course. One of them still had access to their solution, and another one had access to the autograder's solution by virtue of being a TA for the course. The team made the most progress fuzzing those two `malloc` implementations.

For the test harness, the team discovered a program written by Professor Emery Berger of University of Massachusetts Amherst called `hangover` [19]. It was originally designed to fuzz his own memory allocator, but the authors were able to adapt it to their needs. In doing so, they uncovered many bugs in the original source code. For instance, `hangover` populates allocated blocks with known data and checks that the data hasn't been corrupted when it tries to free it. However, the way the data was populated differed from how it was checked, leading to `SIGABRT`s even with correct behavior. For another example, `hangover` used C++'s `auto` when computing known what the aforementioned known data should be, but it used a `char` when reading the data back. This caused otherwise equal bytes to differ, again causing spurious failures. Eventually, all these issues were patched, and the authors submitted a pull request [20].

The authors then ran this patched version of `hangover` through AFL for over a day. The autograder's solution produced no crashes. It only hung twice, and both of those couldn't be reproduced outside of AFL. For the team member's solution, however, fuzzing uncovered a major bug. When adding a block to the end of the free list, the allocator doesn't properly clear the `next` pointer of said block. Since blocks can reuse freed user-controlled memory for metadata, this effectively allows an attacker to smuggle a forged block into the free list. By finding a chunk of data that could emulate heap metadata, a malicious actor could gain access to a block of writable memory through grooming the heap such that they were able to allocate may small fake blocks, finally returning a writable pointer directly to the chunk itself. One way this could be exploited is by setting up the heap to point into the Global Offset Table (GOT). An attacker could then gain access to the GOT, leveraging said access to both defeat the randomization of the C library within system memory and gain arbitrary code execution due to a function pointer overwrite [21].

To demonstrate the severity of this vulnerability, the authors created a simple proof-of-concept (PoC) program with basic functionality, mirroring the "four function heap" programs commonly seen in cybersecurity competitions. As a first step, the authors set up the heap to point into the GOT by finding a block of writable memory located physically right before the GOT. straddling the boundaries between writable and unwritable memory. With this access, the authors decided to first

leak the address of `puts`, a common function in libc, in order to determine libc's base address. As mentioned before, libc's location is randomized on every execution due to Address-Space Layout Randomization, enabled by default in all modern operating systems [21]. With libc located, the authors used `one_gadget` to symbolically solve for a location where a system shell could be opened, then overwrote `abort`, an available function pointer to this address [22]. Finally, all that has to be done is to run the `abort` function, leading to full remote code execution.

As a concluding remark, it's worth noting that the custom allocator from CS 2110 is remarkably similar to the implementation in AVR's libc. Perhaps by studying the potential vulnerabilities here, the team can gain insight into that allocator's attack surface.

VI. Conclusion

This paper set out to investigate techniques for automated vulnerability auditing when applied in an embedded systems concept. Specifically, the authors investigated the application of fuzzing techniques to embedded environments like AVR Libc, musl, and a custom memory allocator written by a team member. While efforts were put into all of these research targets, the custom memory allocator proved to be the most fruitful, leading the team to redirect their efforts there. After some fuzzing, the teams identified a critical bug in the custom memory allocator, then created a proof-of-concept exploit that uses this vulnerability to gain remote code execution on a target machine. Finally, the authors speculate on the applicability of these techniques to more common embedded targets like AVR libc.

References

[1] P. Godefroid, "A brief introduction to fuzzing and why it's an important tool for developers." [Online]. Available: https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/

[2] C. Cimpanu, "Google's automated fuzz bot has found over 9,000 bugs in the past two years." [Online]. Available: https://www.zdnet.com/article/googles-automated-fuzz-bot-has-found-over-9000-bugs-in-the-past-two-years/

[3] "fuzzing.info." [Online]. Available: https://fuzzinginfo.wordpress.com/history

[4] A. Roldan, "Sudo heap overflow cve-2021-3156." [Online]. Available: https://fluidattacks.com/blog/fuzzing-sudo/

[5] "Mac OS X airport update 2007-001." [Online]. Available: https://vulners.com/nessus/MACOSX_AIRPORT_2007-001.NASL

[6] LiveOverflow, "The heap: How to exploit a heap overflow." [Online]. Available: https://www.youtube.com/watch?v=TLa2VqcGGEQ

[7] A. Jain, "Cve-2021-3156: Heap-based buffer overflow in sudo (baron samedit)." [Online]. Available: https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit

[8] 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 *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, ISOC, Ed., San Diego, 2018, © ISOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA and is available at : http://dx.doi.org/10.14722/NDSS.2018.23166.

[9] M. Pucher, C. Kudera, and G. Merzdovnik, "Avrs: Emulating avr microcontrollers for reverse engineering and security testing," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: http://eprints.cs.univie.ac.at/7092/

[10] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[11] A. Jain, "CVE-2021-3156: Heap-based buffer overflow in `sudo` (Baron Samedit)." [Online]. Available: blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit

[12] LiveOverflow, "The heap: How to exploit a heap overflow." [Online]. Available: youtube.com/watch?v=TfJrU95q1J4

[13] Glitches0and0stuff, "Reach the credits from kokiri forest using ace: Ocarina of Time glitch explained." [Online]. Available: youtube.com/watch?v=wdRJWDKb5Bo

[14] SeedBorn, "How speedrunners warp straight to the moon in majoras mask." [Online]. Available: youtube.com/watch?v=3-Gy4Zw1pGo

[15] S. Tong, "This bug doesn't exist on x86: Exploiting an ARM-only race condition." [Online]. Available: github.com/stong/how-to-exploit-a-double-free

[16] Cyber Security Club @ tOSU, "Mooosl." [Online]. Available: github.com/cscosu/ctf-writeups/tree/master/2021/def_con_quals/mooosl

[17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[18] AFLPlusPlus, "`include/afl-as.h`." [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/include/afl-as.h

[19] E. Berger, "hangover: Basic fuzzer for malloc implementations." [Online]. Available: https://github.com/emeryberger/hangover/

[20] A. Ratnani, Z. Lipschutz, S. Hua, and C. Reid, "Prevent segfaults and add data corruption checks." [Online]. Available: https://github.com/emeryberger/hangover/pull/2

[21] B. Spengler, "Pax: The guaranteed end of arbitrary code execution." [Online]. Available: https://grsecurity.net/PaX-presentation.pdf

[22] D. Chiang, "$\mathrm{One}_g adget : The best tool for finding one gadget rce in libc.so.6.'' [Online]. Available : https://github.com/david942j/one_gadget$