Investigating the Security of the Smaller C Compiler

Anonymous Author(s)

ABSTRACT

The abstract is a fancy way to saying **Summary.** It should preferably be two paragraphs that summarize your report. Abstracts are read independently from the rest of the report so you must not cite your report or any other papers. Study other abstracts in the papers you have been reading to understand what an abstract should mean, although many are poorly written.

The abstract is not an introduction or overview of your report! It is a summary of your report, which should include the background, context, content, and contributions/results of your report. Make sure you only take credit for what you did (which is the comparison and your learning) and mention all work (research ideas, software, etc.) done by others.

The first paragraph should provide an overview of the topics being covered in the report. The second paragraph should describe what you learned and and how it would be meaningful to your reader, but do not this in the first person.

Write the entire abstract in the third person and in past tense. It should typically be around 200-250 words.

KEYWORDS

Come up with your own descriptive keywords to make possible for a potential reader to find your report.

1 OVERVIEW

We are investigating how the "Smaller C" compiler addresses various security vulnerabilities common in compilers, examining the security impact of any bugs or compiler restrictions, and documenting any vulnerabilities that we find.

Some compilers are able to catch and address many of the most common software vulnerabilities described in the Common Weakness Enumeration.[2] Of these vulnerabilities, only a few can theoretically be caught by compilers. The first of these bugs is Out-of-bounds read and writes, which occur when memory outside of an allocated space is accessed in the program. This can lead to undefined behavior, and can be exploited by malicious actors to change program functionality. Ideally, static out-of-bounds reference should be caught by the compiler and flagged as an error, but dynamic accesses can still pass through. A second issue is use after free (UAF), in which memory that was allocated for use is freed, then accessed later. Again, this can be used to cause unintended behavior if exploited. In sequential programs, it should be possible to prevent this, but it would also be ideal to ensure that memory is freed in the same scope it is declared, and only after it has finished being used. Most C compilers do not address this, but using an address sanitation procedure to detect memory errors in implementation can reduce this vulnerability. Another possible vulnerability is null-pointer dereferencing. Null-pointer dereferencing refers to attempting to read memory from a null-pointer, which could be a pointer that has not yet been assigned, or a pointer that was freed and reset. Checks like those described above could help with this, but these tasks will likely be harder, due to the single-pass nature of Smaller C. A final vulnerability that could be addressed is the use of shared resources with improper synchronization. This refers to accessing any shared data or resource in a multi-threaded system in a way that either interferes with other programs trying to use the same resource, or prevents other programs from being able to access that resource. If a resource is declared as "critical," which means that it should not be accessed by multiple sources during an operation, it should always be surrounded by a semaphore, lock, or other equivalent security measure that ensures exclusive access to it. A compiler could check for the entering and exiting of critical sections when accessing critical resources, so we will see if Smaller C implements any such behavior. Over the course of this project, we intend to document how Smaller C addresses these vulnerabilities, and provide solutions or improvements. It may not be possible to prevent these issues, but it should be possible to add safety features to catch these issues most of the time.

A vulnerability that C compilers often have is one where stack data can be overwritten by invalid memory accesses, overflows, or other common code vulnerabilities. In the worst case, overwriting data on the stack can allow the function return address to be overwritten, allowing malicious actors to execute arbitrary code by inserting a faulty address. It will be important to investigate how Smaller C attempts to prevent these types of attacks. David Monniaux details one method for catching these attacks, in which a randomly generated "canary" value is inserted before the location that the return address is stored. [4] This value is checked at the end of function execution. If it was changed, then much like a canary in a coal mine, it signifies that something has gone wrong. In this case, it means that the return address was likely changed too, so the program stops execution to prevent any undesired behavior. This method may not even be possible in a single-pass compiler, so some other work-around may be necessary.

Code Optimization in compilers can often unintentionally remove security checks or make code less secure. This can be especially true when compiling on hardware different than the intended program location. We will explore how Smaller C handles common optimization techniques, such as dead store elimination. D'Silva et al. laid out how dead store elimination in GCC removed a necessary security feature from the function crypt(). [3] We will see if Smaller C has similar erroneous optimization. Another common error with optimization is how the compiler handles undefined behavior within the C standard. There have been noted cases, by Wang et al. where various security checks are removed due to it being undefined by the C standard. [5]

Smaller C has some unique limitations that may, if improperly implemented, lead to security vulnerabilities. First, there is no stack overflow check, and very few compile-time checks compared to most multi-pass compilers. This might cause a program to be able to

be frozen by putting it into an infinite loop. Secondly, large integers constant are ambiguously assigned to types of different sizes unless they are explicitly marked. This can cause different amounts of space to be allocated, which can cause issues in extreme cases.

2 RELATED WORK

Review the related work in the literature, both research and practice, to place your project in perspective, and what other people have been doing to address this problem. Make sure your literature survey is fairly complete and recent, which means you should references that are as new as possible, i.e., some (not necessarily all) should be in the current or last year.

You must cite your sources correctly per ACM style guidelines (and of course, you need to use LaTeXand BibTeXcorrectly).

3 DESIGN AND IMPLEMENTATION

Use this section to describe the basic design, architecture, and implementation of your project

4 ANALYSIS

Use this section to describe the analysis of your project that you conducted, and whether the results are meaningful or not.

Also, discuss what all you learned from the project, especially what mistakes to avoid in the future.

5 LEGAL CONSIDERATIONS

Use this section to discuss legal issues relevant to your project, especially relating aspects of data that are relevant to your project.

Use the textbook and your readings to guide the legal aspects of your discussion. Look at the laws that have been passed in recent years, and look at legislation that is being proposed in the space covered by your project.

6 ETHICAL CONSIDERATIONS

Use this section to discuss ethical issues relevant to your project, especially relating aspects of data that are relevant to your project.

Use the ACM Code to guide the ethical aspects of your discussion [1].

7 CONCLUSIONS

Use this section to describe the current status of your work and what else needs to be done.

Also, discuss what further directions your work can be taken by others.

Finally, present some final words to place your project in perspective.

REFERENCES

[1] Association for Computing Machinery. 2018. ACM Code of Ethics and Professional Conduct. ACM, New York. https://www.acm.org/code-of-ethics.
[2] CWE. 2023. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html
[3] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. IEEE, IEEE, New York, NY, 73–87.
[4] David Monniaux. 2024. Memory Simulations, Security and Optimization in a Verified Compiler. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Association for Computing Machinery, Association for Computing Machinery, London, UK, 103–117.
[5] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, Association for Computing Machinery, New York, NY, 260–275.