Investigating the Security of the Smaller C Compiler

Anonymous Author(s)

ABSTRACT

Programs written in C and C++ can be vulnerable to a wide range of memory safety issues. From out-of-bounds reads and writes to use after free errors, developers have to be careful to ensure that they release code without critical security flaws. Some of the most common vulnerabilities include out-of-bounds (OOB) read and writes, use after free (UAF), and improper use of shared resources. Often, compilers have tools built in that help developers catch these memory safety violations. While it is not possible to catch or prevent all memory safety vulnerabilities in a compiler, especially in languages where there is unsafe behavior explicitly allowed by the standard like C, there are many approaches that can be applied to increase memory safety.

This paper investigates a few methods to catch memory safety violations to see if they could be applied to a single-pass C compiler called "Smaller C". This includes a basic Canary implementation to prevent stack-smashing attacks and an implementation of poisoning memory addresses to catch UAF errors. Along with this, the research covers built-in address sanitation and investigation into compile-time checks for potential race conditions, which are effective, but potentially cumbersome– and by extension difficult to implement in the "Smaller C" compiler with traditional methods. The investigation into security methods that can be used in a variety of compilers also sheds light on what is viable for lighter systems, and what techniques can avoid losing out on hard-earned performance optimizations.

KEYWORDS

Compiler, Security, Single-Pass Compiler, C, C++, Use-After-Free, Out-Of-Bounds, Memory Safety, Stack-Smashing

1 OVERVIEW

Memory safety vulnerabilities are a constant security threat to programs written in C and C++. As recently as this year, major consumer software companies still uncover security vulnerabilities that cause privacy concerns for users. Large tech companies such as Apple, Google, and Microsoft still regularly find seemingly basic vulnerabilities regarding incorrect memory usage [2].

The Common Weakness Enumeration [6] describes many of the most common memory safety issues. Of these vulnerabilities, only a few can theoretically be caught by compilers.

The first of these bugs is out-of-bounds (OOB) 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 references should be caught by the compiler and flagged as an error, but dynamic accesses can still pass through. Some compilers implement a memory tagging procedure to detect OOB errors at runtime [5]. In this procedure, memory blocks are locked with a tag that is stored in the end bits of a pointer. When a memory access is made, the tag in the pointer is checked against the lock on the memory block. If they do not match, that means that the pointer being used is meant to access a different data block, and is trying to access data that it should not be able to access. In this scenario, program execution is terminated to prevent any security breaches.

Memory tagging procedures generally impose a large overhead and can be difficult to implement, so many compilers focus on preventing OOB errors where they are most important: on the stack. 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. 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 [9]. This value is checked at the end of function execution, before a value is returned. If the value 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.

A second issue is use after free (UAF), in which memory that was allocated for use is freed, then accessed later. This can be used to cause undefined behavior when 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 [10]. 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. Since there is only a single pass through each compilation unit, and no intermediate steps or representation, there are potential security flaws caused by one portion of a non-externalized unit that has emergent insecure behavior due to another subsequent portion. This means that the source code may need to be reviewed semantically for behaviors normally caught through the use of comparative or sequential compilation.

```
1  int useAfterFree(int count)
2  {
3      printf("Use After Free\n");
4
5      // Allocate a buffer on the heap
6      int* someInts = calloc(intSize, count);
7      for (int i = 0; i < count; i++)
8          someInts[i] = 8 * i;
9
10     // Check what's on the heap
11     for (int i = 0; i < count; ++i)
12         printf("\t\t%d\n", someInts[i]);
13     printf("\n");
14
```

```
15      // Free the buffer
16      free(someInts);
17
18      // This is our data after freeing
19      for (int i = 0; i < count; ++i)
20          printf("\t\t%d\n", someInts[i]);
21      printf("\n");
22  }
```

useafterfree.c

A particularly difficult to diagnose 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 unexpected behavior when entering and exiting of critical sections of code [8], so we will see if Smaller C implements any such behavior.

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 integer constants 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.

This paper documents security issues present in Smaller C and puts forward a number of improvements that will reduce the vulnerabilities present in code compiled with Smaller C.

2 RELATED WORK

2.1 Smaller C function

It is important to understand the steps taken by Smaller C to compile code. Before any compilation happens, there is typically a preprocessing step. In GCC, the preprocessor performs a few simple tasks, such as breaking the program into lines, removing comments, and merging continued lines [1]. Ultimately, this step tends to not have much impact on the finished code, as long as it functions properly. The code for Smaller C's preprocessor was not included in the GitHub, only the executable files.

In the compilation step, a traditional compiler might begin by using a lexical analyzer to divide the characters in the program into tokens, which are distinct symbols with meaning in the program. This includes math symbols, semicolons, file names, and all other atomic elements of code. From there, this stream of tokens would be syntactically analyzed, verifying that the tokens appear in a valid order. Then, the function of the tokens would be determined using a semantic analyzer, and an intermediary code file would be generated. This intermediary code is more suited to optimization and modification. The optimization step involves reordering code blocks and adjusting code so that it runs faster. Finally, machine code is generated. Most compilers perform each step in sequence, not moving onto the next until the previous has been completed. In contrast, Smaller C parses each token, verifies it as being properly syntactically ordered and adds it to a block of symbols.

As soon as a block of code can be understood, it is immediately converted into machine code. This means that any improvements made to Smaller C cannot come from code analysis, since there is no period of time in which more than a single line of code is analyzed. As such, improvements to Smaller C have to be made primarily in the code generation step, padding unsafe actions written in C into memory safe interactions in machine code.

2.2 Code Optimization

Code Optimization in compilers often unintentionally removes security checks and makes code less secure than the unoptimized variant. 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()` [7].

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 [11]. After looking into the codebase of Smaller C, no evidence of code optimization was found, although it seems as though an intermediary called RetroBSD may reorder some instructions. Assembly code is generated based on the C program input, with very little in the way of modifications that could create a security threat.

2.3 Related Memory Security Improvements

This subsection discusses several memory safety improvements that have been introduced in other compilers that likely will not work in Smaller C.

Memory tagging, like described above, is a process in which sections of memory are locked so that they can only be accessed by certain pointers. Chen et al. describes an implementation that works to secure memory on the heap [5].

During compilation, the author's tool, HeMate, finds the type, base address, and size, and uses it to generate tags for each allocated block of memory. HeMate then stores the tags in the ending bits of pointers such that they contain type information, as well as a randomly generated unique tag. This requires every memory allocation to have a concrete type at compile time, which, despite being a standard style for C, is not assured by the Smaller C compiler. This requirement makes storing types in memory tags all but impossible in Smaller C. Unfortunately, memory tagging in general does not seem to be possible either. While it would be relatively easy to add a lock to a block of memory with a simple modification to code generation, editing the pointers to include a tag would require a significant refactoring of Smaller C. Since it is a single pass system as well, it would also be extremely difficult to ensure that pointers to the same location in memory have the same tag, as is typical in memory tagging procedures. Memory tagging does not seem to be feasible in Smaller C.

3 DESIGN AND IMPLEMENTATION

The first fix proposed for the Smaller C compiler is to add Canaries to protect against Stack Smashing. Currently, Smaller C has no such protections, and allows a sample program?? to compile and run in such a manner that it corrupts the stack during execution. To prevent this, the code generation section of Smaller C can be modified to include a canary check. In the `GenFxnProlog` function, which sets up the stack frame for a function, assembly instructions could be written to add a canary to the end of the stack frame. In the function `GenFxnEpilog`, which handles the ending of function execution, assembly instructions could be added into the code to check that canary. If the value differs, a branch error is thrown and the program stops execution. A difficulty with this approach is considering where to store the canary outside of the stack frame, so that it can be checked. Originally, a sensible approach seemed to be storing it inside the stack frame of the previous function, but this would not prevent stack smashing attacks. To check this canary, we would have to return to the previous stack frame, which is impossible if the return address was overwritten in a stack smashing attack. Another idea is to store this value in a register, but Smaller C has no spare registers that are guaranteed to be unmodified by function execution, so this cannot be done. Another approach is needed to implement canaries in Smaller C.

```
1  void GenFxnEpilog(void) {
2    GenUpdateFrameSize();
3    // Check if the function called another function
4    if (!GenLeaf) {
5      // Load Canary from Stack into temp register 1
6      GenPrintInstr2Operands(
7        MipsInstrLW, 0,
8        MipsOpRegT1, 0,
9        MipsOpRegFp, 12
10     );
11     // Load Canary from Storage Location into
           temp register 2
12     GenPrintInstr2Operands(
13       MipsInstrLW, 0,
14       MipsOpRegT2, 0,
15       CanaryStorageLocation, 0
16     );
17     // Compare the two values, if they aren't
           equal, crash.
18     GenPrintInstr3Operands(
19       MipsInstrBNE, 0,
20       MipsOpRegT1, 0,
21       MipsOpRegT2, 0,
22       MipsInstrExit, 0
23     );
24     // Load the return address we stored on the
           stack into the return address register
25     GenPrintInstr2Operands(
26       MipsInstrLW, 0,
27       MipsOpRegRa, 0,
28       MipsOpIndRegFp, 4
29     );
30   }
31   // Load the previous frame pointer back into a
         register
32   GenPrintInstr2Operands(
33     MipsInstrLW, 0,
34     MipsOpRegFp, 0,
35     MipsOpIndRegFp, 0
36   );
37   // Move the stack pointer past the function
         frame
38   GenPrintInstr3Operands(
39     MipsInstrAddU, 0,
40     MipsOpRegSp, 0,
41     MipsOpRegSp, 0,
42     MipsOpConst, 12 - CurFxnMinLocalOfs
43   );  //12 = RA + FP + Canary
44   // Jump to the return address and continue
         execution
45   GenPrintInstr1Operand(
46     MipsInstrJ, 0,
47     MipsOpRegRa, 0
48   );
49 }
```

genfxnepilog.c

The next fix proposed for the Smaller C compiler is adding memory address sanitization. This is a method of checking code on compile time for errors like buffer overflow and UAF. This is generally done by adding marks to memory addresses. These are either adding 32 byte aligned poisoned "redzones" before and after contiguous areas of memory, and poison tags when a memory address has been freed [3]. The redzone allows checks to be made if any access to memory is in one of these zones, and an error can be thrown and the compiler stopped. The poison tags for freeing equate to an if statement before any dereference of memory to check if that address has been poisoned. Both of these implementations heavily make use of the ShadowStack [10].

To implement these poison checks for Smaller C there need to be changes made to the `ParseDecl` and `derefAnyPtr` functions, as well as creating a list of poisoned memory addresses. After line 8368, where a function name is caught, there can be a check to see if it is freed, and add the memory address in its parameters to the list of poisoned memory locations. A check can also be made in the `derefAnyPtr` function to see if the given memory address is poisoned, and error out appropriately.

```
1  char* identGlobal = NULL;
2  char* poisonedIdents[1024];
3  int poisonedIndex = 0;
4  char prevUnaryStar = 0;
5
6  ...
7
8  char toPoison = 0;
9  if(identGlobal != NULL && strcmp(identGlobal,
       "free") == 0) {
10   toPoison = 1;
11 }
```

```
12
13 // DONE: support __func__
14 char* ident = IdentTable + s;
15 if(toPoison) {
16   poisonedIdents[poisonedIndex] = ident;
17   poisonedIndex++;
18 }
19
20 if(prevUnaryStar) {
21   for(int i = 0; i < poisonedIndex; i++) {
22     if(strcmp(ident, poisonedIdents[i]) == 0) {
23       error("ident: %s is poisoned\n", ident);
24     }
25   }
26 }
27
28 identGlobal = ident;
```

asanpoison.c

4 ANALYSIS

4.1 Smaller C Compiler Memory Safety Issues

While examining Smaller C, we were able to document a number of other security vulnerabilities present.

Firstly, there are a number of errors present in the compiler, where expressions that should be syntactically or semantically invalid are compiled and able to be executed. One of these issues is related to the fact that the compiler treats structs as lvalues, which allows code to compile where structures are being used to assign to arrays of structures or function pointers. This should cause crashes on its own, and definitely can lead to unintended behaviors in programs. Another bug is that extern and static variables can have the same name, which should not be allowed since it produces ambiguous behavior due to shadowing.

One concerning property of the Smaller C compiler is that it does not check for pointer types when dereferencing, it just checks that the types have the same size. Technically, this should not lead to any memory safety issues on its own, but it could cause unintended behavior, which could lead to a memory safety violation.

Bugs related to improper synchronization and race conditions that are baked into programs can often be caught through the use of techniques that test the code using several optimization levels and monitoring critical sections of code [8]. However, since Smaller C uses one-pass compilation, it isn't possible to do any comparative or sequential analysis of various sections of code together, since the compile units are differentiated and converted to machine code procedurally. In our investigation we have found no way of mitigating a race condition in any way in a single pass, save for potentially scanning code semantically to attempt to find a race– but this would produce at best a warning.

5 LEGAL CONSIDERATIONS

TBD- looking into legal implications of compilers. Will discuss legal implications of releasing insecure code, but want to also tie this back into compilers

6 ETHICAL CONSIDERATIONS

The ACM code of ethics has a few relevant sections when it comes to memory safety vulnerabilities in compilers [4].

Code 1.2: "Avoid harm" is relevant to the designer of the compiler, as well as software developers who use it. Memory security vulnerabilities can cause software to stop working suddenly, which could cause loss of life if a hospital suddenly was unable to care for patients. Additionally, hackers can take control of programs with memory security vulnerabilities. This could be dangerous, if for example, they got control of an application that controlled a defensive military installation and prevented it from intercepting missiles. As such, a compiler writer wants to ensure that their compiler will prevent memory vulnerabilities, and developers want to write code that prevents them.

Another relevant section is Code 1.3: "Be honest and trustworthy." A compiler should not misrepresent the extent to which it prevents or detects memory safety vulnerabilities, or a developer might unintentionally release a program that is unsafe.

On the same note, Code 2.7: "Foster public awareness and understanding of computing, related technologies, and their consequences" requires that a compiler vendor explain the importance of vulnerabilities that may not be caught, and should also put forward methods to prevent issues that could make it through to compile time.

Code 2.9: "Design and implement systems that are robustly and usably secure" is also relevant. As discussed previously, some compilers may optimize code in a way that introduces security vulnerabilities, which violates this principle. Developers should also be wary of using compilers that do not guarantee memory safety, since releasing any insecure product is a violation of this code.

Code 3.7: "Recognize and take special care of systems that become integrated into the infrastructure of society" can be relevant for any compilers that aim to be used by any IDE. It is the responsibility of large developer tools producers to ship compilers that come packaged with their code environments, such as Visual Studio by Microsoft, that are memory safe by default, since many developers use that compiler without ever thinking about it.

Since a compiler is a tool used by other developers, creating an insecure compiler can have a knock on effect that results in widespread vulnerabilities. Thus, it is very important for compilers to enforce memory safety wherever possible as an ethical duty to the developers who use them.

7 CONCLUSIONS

Reading over and understanding a program like Smaller C, in which all functions are static and have hundreds of side effects is difficult. As such, a majority of progress up until this point has been spent analyzing the program execution and figuring out how everything interacts. In the future, we intend to implement more programs to show vulnerabilities that pass, and finish implementing a few fixes to reduce the prevalence of memory vulnerabilities in Smaller C.

We do not recommend using Smaller C to compile any program intended for serious use: Smaller C was a hobby project and is not able to perform any of the security or optimization tasks that we expect a compiler to do.

REFERENCES

[1] [n. d.]. *GNU Manual: CPP.* https://gcc.gnu.org/onlinedocs/cpp/

[2] 2024. *About the security content of iOS 18 and iPadOS 18.* https://support.apple.com/en-us/121250

[3] Mitch Phillips Alexander Potapenko. 2017. *AddressSanitizerAlgorithm.* https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

[4] Association for Computing Machinery. 2018. ACM Code of Ethics and Professional Conduct. ACM, New York. https://www.acm.org/code-of-ethics.

[5] Yu-Chang Chen and Shih-Wei Li. 2024. HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension. In *Proceedings of the 19th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '24)*. Association for Computing Machinery, New York, NY, USA, Article 30, 11 pages. https://doi.org/10.1145/3664476.3664492

[6] CWE. 2023. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[7] 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.

[8] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 238–251. https://doi.org/10.1145/3582016.3582053

[9] 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.

[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[11] 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.