Stack Faults: Memory Access Permissions for Stack Frames

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Ratik Mathur

Spring, 2024

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Rosanne Vrugtman, Department of Computer Science

Stack Faults: Memory Access Permissions for Stack Frames

CS4991 Capstone Report, 2024

Ratik Mathur
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
rdm7rkm@virginia.edu

ABSTRACT

Buffer overflows are a common security vulnerability that plague many code bases, notably those written in C/C++, by writing data into a buffer past its capacity which may replace important data on the stack. I propose a stack frame-level memory permission check for data on the stack, inspired by page faults that are process-level memory permission checks. When a function call is made, an identifier for the new frame is pushed onto the stack; this identifier is checked whenever we write to the stack, and if a change in identifier is detected, then the program "faults" and a memory permission error (such as "Segmentation Fault") is displayed. This would serve as a method of mitigating Buffer Overflow attacks with minimal performance degradation, ideally similar degradation to bounds checking on the buffer. Future work would include extending the idea of numbering frames to get much faster than bounds checking (maybe through the hardware), as well as expanding this concept of faulting the program to other areas that are vulnerable to unauthorized memory access.

1. INTRODUCTION

Older languages such as C and C++ are still relevant for today's developers. These programming languages work rather closely to the underlying hardware and provide a convenient abstraction for said hardware. This means that they require the developer to tell the computer exactly what to do and thus make the developer responsible for securing their code properly to prevent vulnerabilities. A notable vulnerability with this model is the "buffer overflow" attack. Buffer overflows occur when the amount of data written to a location (the buffer) exceeds the storage space the computer has allocated for that data (overflowing it). For example, if a program wants to take the user's name as input and then display it to the user, the developer might allocate ten bytes on the server to store that name. However, unless extra precautions are taken, a user of the program could enter a megabyte of data. This would all get written to the server, writing well past the allocated ten bytes, overwriting important data in the process.

All major programming languages allow placement of lines of codes into a "function" that can then be referenced (or "called") by the name given to it. This simultaneously introduces abstraction and allows for code to be reused throughout a program. Every time one function calls another, as the called function is likely in a different section of memory, the computer must make sure to save all important data about the calling function before "jumping" to the memory location where the desired function is. This data is placed into what is called a "stack frame" within a part of the computer's memory called the "stack" **[1]**. Notable among this data is the caller's "return address," which specifies where in memory

the computer should go back to once it has finished the function.

Buffer overflows, like the example mentioned above, work by overwriting the data in other stack frames. If an attacker can determine how the stack frames are laid out, they could attempt to overflow exactly enough data that they overwrite the return address with malware that opens a backdoor. When a function finishes executing, the computer will look at whatever data is in the slot the caller's return address should be in, unconditionally jump to that memory address, and begin executing any code it locates there.

One of the first such exploits was used by the "Morris worm" which overwrote the return address to point to code that opened up a reverse shell (a command line interface) **[2], [3]**.

To defend against such attacks, developers either need to add additional instructions to properly check the data before writing it to a buffer or use a language (compiler) that will insert these instructions for them. This slows the application down as there is literally more code to be run.

However, while the return address should never be modified by a buffer write, it is rather common for functions to modify other data (including other buffers) from different stack frames, typically through a process called pass-by-reference. A simple example would be an update_username function, which should modify the memory address at which the username lives, which would be in a previous frame.

2. RELATED WORKS

This idea is like the concept of "Page Faults" which effectively does this but for entire programs: they prevent other programs from touching memory they should not have. I propose "Stack Faults" that crash the program if a stack frame modifies memory it should not have.

While I have not found a paper attempting to number frames as I am proposing, there has been numerous works in buffer overflow mitigation that specifically try to crash the program when writing across frames. Such examples are stack canaries, guard pages, stack checksums, shadow stacks, etc.

An example of a canary is in the "StackGuard" paper. StackGuard places a secret word next to the return address and checks if that word has been modified, which would indicate that the return address has been modified. It also can prevent writes in the first place by make the return address read only while the program is executing, although this is slower than a canary **[4]**.

There have also been security analyses conducted confirming that certain implementations of shadow stacks are an effective and performant mitigation technique. Shadow stacks operate by making a copy of the return addresses and storing them elsewhere in memory. These copies are then used to verify the integrity of the stack when returning from a function call **[5]**.

3. PROPOSAL DESIGN

To mitigate these vulnerabilities, I propose increasing the granularity of permissions for data within stack frames. If a buffer overflow attempts to overwrite the return address, which is in the previous stack frame, it will fail to do so as it does not have the permissions. More specifically, the hardware will crash the entire program since the program attempted to access invalid memory.

Since we want to make sure the program does not crash in valid pass-by-reference scenarios, we should specifically be concerning ourselves with the scenario of writing **across** stack frames. That is, beginning to write to a buffer in one stack frame and then end up writing to memory in another stack frame. This way, valid use cases for passing by reference will be allowed

as the writing would occur within the appropriate frame.

To achieve this, I will be modifying the source code for an operating system to keep track of which frame is being written to anytime data is placed on the stack. If a change in frame is detected, this would almost certainly be due to a buffer overflow as the stack frames should have adequate space allocated in them to support all the data a buffer would require. I will edit the code to send a signal to the operating system, referred to as "interrupting" the OS, to crash the current process when a change in stack frame is detected.

4. ANTICIPATED RESULTS

I anticipate that this approach to mitigating Buffer Overflows will incur minimal performance degradation and that the performance will be comparable to bounds checking. For this, I propose creating benchmarks that push a lot of buffers onto the stack and then creating a second version of the benchmarks that perform bounds checking; the first set would be run with my proposed mitigation strategy and the second would be run without, allowing a performance comparison between the two. For dealing with optimizations that would place data into registers over the stack, we can either disable the optimization or create six dummy variables before the buffers to force them onto the stack as compiler optimizations will often place up to six variables into registers.

5. CONCLUSION

I propose a mitigation strategy for buffer overflow vulnerabilities that prevents programs from executing if they are found to have been poisoned by such an attack. It accomplishes this by numbering frames and then checking if a single write instruction crosses multiple frames. Rather than preventing buffer overflows from taking place, as is commonly seen in various modern programming languages, it instead focuses on blocking programs that have been compromised; this is an approach that is commonly seen in the literature (such as canaries and shadow stacks) for mitigating these vulnerabilities. Inspired by prior works, it is estimated to have minimal performance degradation allowing it to serve as a viable alternative to other strategies.

6. FUTURE WORK

This project will serve as a feasibility study, showing that numbering frames, or even looking at stack frame permissions in general, is a worthwhile area of research to pursue. There does not seem to be much literature on managing access permissions at the granularity of stack frames in general. Another idea I have is to copy the idea of page faults quite literally by having metadata in the base pointer for an array to do a "stack table" traversal.

This paper also, in general, uses a data check to crash the program. An alternative would be to trigger interrupts from hardware as page faults often do. The numbering stack frames approach could be implemented similar to page faults but with much less complexity.

REFERENCES

[1] "Stack frames." Accessed: Mar. 21, 2024. [Online]. Available: https://people.cs.rutgers.edu/~pxk/419/notes/frames.html

[2] H. Orman, "The Morris worm: a fifteen-year perspective," *IEEE Secur. Priv.*, vol. 1, no. 5, pp. 35–43, Sep. 2003, doi: 10.1109/MSECP.2003.1236233.

[3] E. H. Spafford, "The internet worm program: an analysis," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, pp. 17–57, Jan. 1989, doi: 10.1145/66093.66095.

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke, “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”.

[5] N. Burow, X. Zhang, and M. Payer, “SoK: Shining Light on Shadow Stacks,” in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2019, pp. 985–999. doi: 10.1109/SP.2019.00076.