

A Simple Machine Simulator for Teaching Stack Frames

Dino Schweitzer Jeff Boleng
United States Air Force Academy
USAFA, CO 80840
+1-719-333-3945

dino.schweitzer@usafa.edu

ABSTRACT

Stack frames are a fundamental concept in computer science often taught in an operating systems or an assembly language programming course. Computer security courses also rely on an understanding of stack frame concepts when teaching buffer overflow attacks. To assist students in learning the fundamentals of stack frames and related concepts, we have developed an interactive Simple Machine Simulator tool that allows students to step through a C-like language program and watch how memory changes. We have used this tool successfully in various courses to teach different aspects of stack frames and their use. This paper will describe the tool, how it is used to teach stack frame concepts, our experience, and future plans.

Categories and Subject Descriptors

K.3.2 [Computer and Information Science Education]: Computer Science education.

General Terms

Experimentation, Languages.

Keywords

Active Learning, Stack Frames, Buffer Overflow, Visualization.

1. INTRODUCTION

Stack frames, or activation records, are a fundamental concept in computer science education. Students need to understand the purpose and operation of stack frames when subroutines are called to understand how parameters are passed, how re-entrant code is enabled, and how recursion is possible. When teaching computer security concepts, buffer overflows most commonly attack a system by “smashing the stack” and inserting malicious executable code directly onto the stack. To understand the threat and details of the attack, students need to understand how stack frames work, and how they can be manipulated to achieve the desired results. In the ACM Computer Science Curriculum 2008, the notion of stack frames, activation records, and subroutine calls appears in multiple places within the curriculum including Programming Fundamentals, Operating Systems, Programming Languages, and Architecture [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.
Copyright 2010 ACM 978-1-60558-885-8/10/03...\$10.00.

At the United States Air Force Academy (USAFA), the basic concepts of stack frames are introduced in various courses and at different levels. Our Computer Organization and Architecture course introduces stack frames at the assembly language programming level to explain the process of passing parameters and a return addresses to subroutines. In the Operating Systems course, stack frames are used to demonstrate process control, concurrency, and reinforce the memory hierarchy introduced in the Computer Architecture course. Finally, in the introductory Computer Security course, stack frames are used in the discussion of buffer overflows.

Unfortunately, in the security course, some of the students are not computer science majors, have never seen stack frames or assembly language, and struggle with the concepts. As a result, in previous offerings, an inordinate amount of time was spent reviewing stack frames rather than focusing on the buffer overflow issue. In addition, we had students performing a hands-on lab in which they intentionally generated a buffer overflow in C, and viewed the results using disassemblers and debuggers. Needless to say, our non-CS majors were confused. To assist in student comprehension and minimize the need to understand additional background material, we developed a Simple Machine Simulator (SMS) interactive tool to demonstrate stack frames and buffer overflows in a hands-on “web lab” environment. Based on the success of the tool, other courses in the major are now using it to introduce stack frame concepts to students in an easy-to-understand context.

The remainder of this paper will describe other approaches found in the literature (Section 2), describe the tool itself (Section 3), demonstrate how the tool was used to teach buffer overflows (Section 4), and conclude with our experience and future plans for the tool (Section 5).

2. OTHER APPROACHES

Visualizing machine registers and internal CPU state is not new. A wide variety of assembly language simulators are available. For instance, SPIM [2] and MIPSTER [3] are both well known and widely used. We use both in our Computer Architecture course. Another related tool is CPU SIM [4].

A specific approach to generating buffer overflow via return address redirection is described in Phillips and Tan [5]. We use a simplified version of this technique in our Computer Security course with the web lab described below. Barnett [6] introduces a scaled down assembly language and assembler used to reduce the required initial knowledge that has influenced our approach. Cribb [7] also describes a similar simulator with a greatly simplified instruction set. Finally, Sundararaman and Back [9] present a more general purpose memory and data structure visualization technique.

Our approach is different from previously presented tools in the following ways:

1. A common goal of the above simulators is to use them to teach assembly language syntax and programming skills. One of our specific goals was to avoid the overhead and complexity of teaching syntax. Our tool is used to illustrate the concept of stack frames and buffer overflows in only two lessons. By greatly simplifying the instruction set and using a very common language syntax (i.e. C like) our simulator is accessible to students that may never have seen assembly language before. In fact, we have successfully used the simulator in a web lab with non computer science majors and received positive feedback.
2. Another common goal of assembly language or machine simulators is to aid in the teaching of hardware concepts such as memory access modes, cache hierarchies, pipelining, etc. This is not our goal. So again, greatly simplifying the simulator and syntax was possible. We represent memory as a linear array of storage. This is sufficient for our purposes to represent how data can be stored in memory, and how function calls and returns result in program counter changes.
3. Many existing simulators are also developed and used to illustrate the relationship and translation of language statements to assembly instructions, then to binary, and lastly to how the binary flows through the internals of the CPU's logic gates. This detailed flow is important to support a computer organization and architecture course, but is a lower level of abstraction than we require.

Our focus is at a higher level of abstraction than existing simulators. We do not require students to understand the intricacies of special purpose registers or memory layouts common in modern processors. Our simulator implements the minimum amount of detail required to teach the specific concepts of stack frames, function calls, and buffer overflows.

A similar effort to create visualizations for use in teaching security concepts was accomplished by Susan Gerhart at Embry Riddle who created a series of playback animations and applets for specifically demonstrating buffer overflow attacks [8]. These demonstrations along with other security-related visualizations are available at <http://nsfsecurity.pr.erau.edu/bom>. Some of these tools are similar to our approach. The user is able to interactively stepwise execute a program and see the effects in a visual memory representation. The user can affect the execution by means of inputting different values at prompts. The tools are designed as applets that can be embedded in web pages. The main differences between these tools and our simulator are:

- Memory is limited to 256 locations that uses primarily color coding to represent types of data as opposed to the specific binary contents.
- Individual instructions are not represented in memory.
- Our simulator provides explicit information on the contents of stack memory to teach students stack frames.
- Our simulator represents a von Neumann architecture with instructions and data explicitly sharing memory.

This model can be used for a variety of educational applications beyond stack frames and buffer overflows.

3. SIMPLE MACHINE SIMULATOR

3.1 Design Goals

The overall purpose of the SMS is to demonstrate the stepwise execution of a pseudo-high level program and see how memory changes to illustrate various concepts. The original focus of the tool was on stack frame concepts, so lower level assembly language details such as addressing modes, registers, and variable length instructions were avoided. To provide this focus, the language needs to be limited, simple, and similar to languages with which students are already familiar. This avoids the need to learn a totally new language in order to use the tool. There is also a requirement for a highly interactive environment in which students can single step through a program, but at any point in time, stop and back up to fully understand what just happened. This allows for "what if" scenarios and student experimentation. The running of the tool needs to be relatively self-evident with minimal instruction on how to make things happen. Our experience with similar tools is that students want to "get going" and not spend time reading instructions or understanding complex interfaces. Finally, it is desirable to embed the tool in customized web pages so that different scenarios and experiments can be presented to the student using the same tool. Sufficient explanatory material can be placed on the web page for the student to complete the exercises without additional references.

3.2 Implementation

Similar to the work at Embry Riddle, SMS is written as a Java applet that can be embedded in web pages along with text for proposed experiments or descriptions. Figure 1 shows the basic layout of the tool. The code pane on the left side of the window displays the current program loaded in memory and highlights the current instruction to execute. The buttons below the code pane control stepwise execution of the program. Below the buttons are the input console and the output pane for performing input and output operations. The right pane displays the contents of memory, and can be viewed in decimal, hex, binary, or character format. Memory consists of 4,096 16-bit locations. The user can scroll through and view all of memory at any point. The Note field in the memory display provides additional explanatory text about what is contained at that particular memory location. A dropdown Help Menu is also present allowing students to display information on instruction formats and how to execute programs.

One of the goals of the tool is to keep the language simple and at a high-level for understandability. To understand the relationship of code to memory, it is necessary for each high-level instruction to have a binary representation in memory. To maintain simplicity, SMS has only two basic data types (integers and strings) and seven basic high-level instruction types:

Assignment
Input
Output
Call subroutine
Return from subroutine
End of Program
NOP

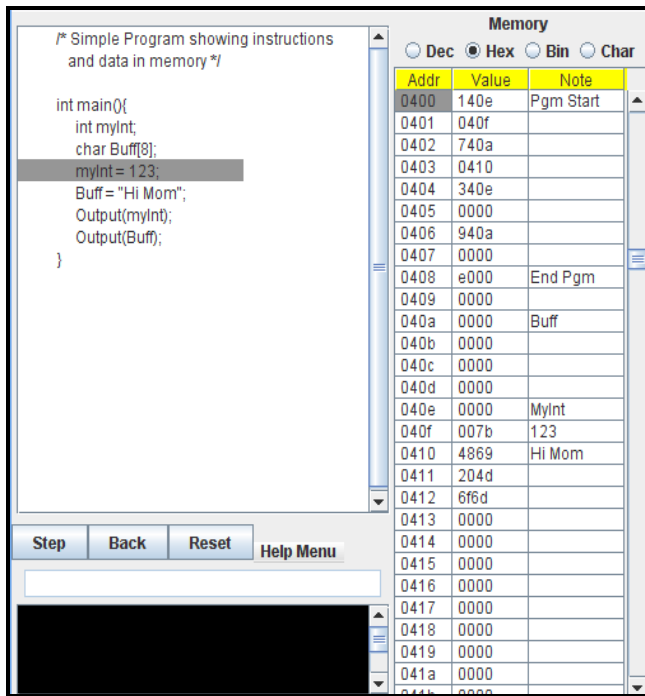


Figure 1. Simple Machine Simulator screen.

The Assignment, Input, and Output instructions can be for either integers or strings. Integers take one memory location (16 bits) and strings are variable length, two characters per word and null terminated. Each instruction takes two words of memory consisting of an op code, and up to two parameters. The source code is designed to look like C with variable declarations and similar syntax. As an example of how the source code translates to memory, Figure 1 shows the instruction “myInt = 123” highlighted in the code pane. The corresponding memory location, at address x0400, is highlighted in the memory pane. The first memory word has four bits representing the op code, which is 1 for integer assignment. The remaining 12 bits are operand 1, which in this example is x40e which is the memory location for myInt. The lower 12 bits of the second word is the second operand, which in this example is x40f. Since our simple language does not have literals, it is necessary to store literal values, 123 in this case, at separate memory locations. The second operand refers to the memory location containing 123.

When the sample program in Figure 1 is executed, the values 123 and “Hi Mom” are printed to the output pane and an “*** END OF PROGRAM ***” message is output when the End of Program instruction at location x408 is executed. As the user steps through the program, the next instruction to execute and corresponding memory location are highlighted to follow what is happening.

In order to use SMS to demonstrate stack frames, it was necessary to add a relative addressing mode. The high order bit of the second instruction word is used to indicate an address relative to the Stack Pointer (SP). When this bit is set, the final address of the operand is calculated by adding the value in the lower 12 bits to the current value for the SP. While more complicated than desirable, this mode is necessary. To help students in

understanding it when a program is executing, the current memory location of the SP is identified in the Note portion of the memory display.

3.3 Executing Programs

The programs in SMS are normally predefined by the instructor. When the tool comes up on a web page, the appropriate program is loaded based on the concepts being presented. The student can execute programs using the Step, Back, and Reset buttons at the bottom of the code pane. Students cannot directly change the high level program during the lab, but they can affect the contents of memory based on input values entered. It is very possible, and, in fact, encouraged in certain labs for students to change the contents of instructions in memory. If the students change an instruction to an illegal instruction and attempt to execute it, they will receive a “run time” error message on the console. This occurs frequently when students first attempt to change the return address from a subroutine call and send the program counter to some arbitrary location in memory.

3.4 Calling Subroutines

To maintain the design goal of simplicity, subroutines are called with either zero or one parameters. The second operand of the call instruction is the parameter. When a subroutine is called, the appropriate information is pushed onto the stack. Specifically, the items pushed are:

- return address
- parameter is present
- local variables

This information composes the stack frame, or activation record, associated with the call. In order for the simulator to know how much stack memory to set aside for local variables, the first word of the subroutine is the count of how many positions on the stack to reserve. Figure 2 shows the SMS state after a subroutine call has been executed. The memory locations between the SP, shown in the Notes field, and the address for the start of the routine comprise the stack frame for this call. In this example, the return address, x404, was the first item on the stack, followed by the parameter, a single word integer value, followed by three words set aside for local variables. The SP grows toward low memory. As the user steps through the statements in the subroutine, any changes to local variables are reflected in the stack memory locations. The high order bit of operands associated with local variables is set indicating addressing relative to the SP. Thus, the second operand for the highlighted input instruction at x03f6 is x8000 indicating it will go at the location pointed to by SP plus a zero offset. When the student executes the statement, they will see the appropriate memory location change.

Figure 3 shows the SMS state after the second subroutine is called from the first. There are now two stack frames on the stack reflecting the return addresses, parameters, and local variables for both calls. As the user steps through the execution, they see the SP dynamically change and can verify the meaning of each value on the stack with the assistance of the notes. Different scenarios can be presented to students with subroutines calling other subroutines or calling themselves (recursion). As the student steps through the execution, they can observe the stack growing and shrinking with each call and return.

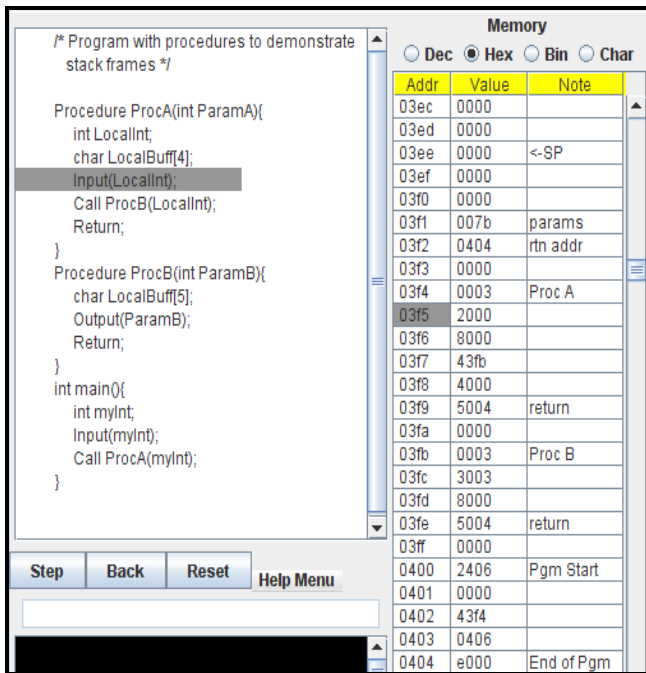


Figure 2. SMS state after subroutine call.

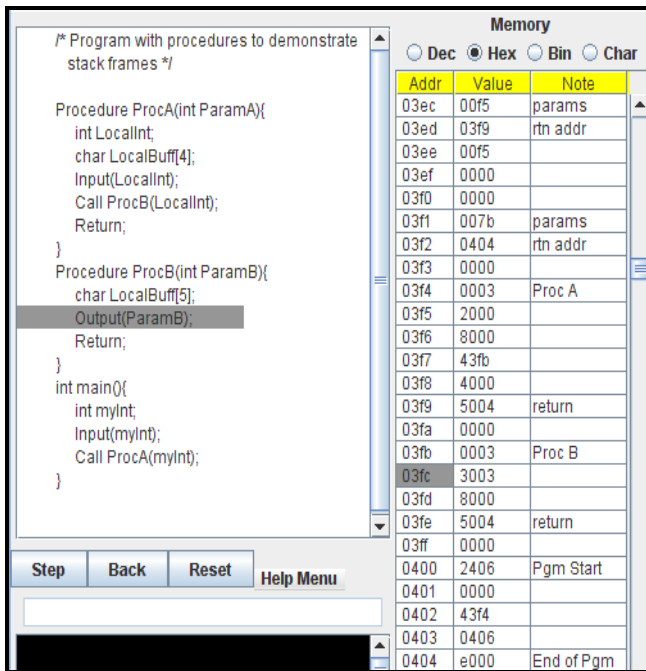


Figure 3. SMS state after second subroutine call.

4. TEACHING BUFFER OVERFLOWS

In the introductory computer security course at USAFA, students receive hands-on training in current tools and techniques for both offensive and defensive aspects of security. Buffer overflows are a primary vulnerability exploited by current attacks and it is important for students to understand what they are, how they are exploited, and how to prevent them. In previous offerings of the course, students conducted a hands-on experiment in which they

generated a buffer overflow in a C program. To observe the overflow, students used code debuggers to step through the program, setting breakpoints, and attempting to manipulate stack content based on carefully crafted input strings. To place code on the stack, students wrote Intel assembly code to accomplish some simple task, assembled the code to determine the machine code equivalent of the program, and finally crafted a string input that would place the appropriate bytes on the stack. The process was involved and required three lessons of class time to explain and step students through the process.

In spring 2009, the course was changed to use SMS to teach buffer overflow. Initially, students were provided the simple example shown in Figure 1 to familiarize themselves with the tool and its operation. Next they were presented with a simple buffer overflow situation shown in Figure 4. Four words of memory are set aside for both Buff1 and Buff2 since they were declared as containing up to eight characters. In the instructions for this example, students are encouraged to enter string values greater than eight characters to see how the input is corrupting subsequent memory locations and affecting program execution. This demonstrates the basic concept of buffer overflow and how it can adversely affect program execution.

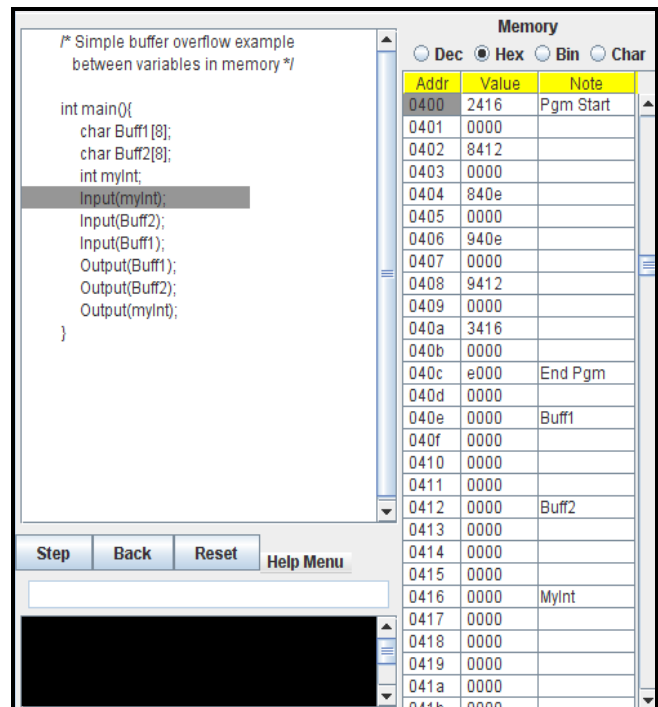


Figure 4. Simple buffer overflow example.

After this example, students were provided the example shown in Figures 2 and 3 along with an explanation of stack frames. This allowed them to re-familiarize themselves (or in some cases see for the first time) how stack frames work and the specific stack contents in this simulator. Next, students were introduced to the concept of “smashing the stack” by stepping through an example in which they input values for a local subroutine variable in an attempt to change the return address on the stack. The goal was to skip the instruction in the main program immediately following the call. This example resulted in many students experiencing the “Illegal Instruction” error code when they jumped to some

random location in memory. The final code sample and exercise was more complicated and led the students through the process of inserting code on the stack through a buffer overflow in order to execute arbitrary code. Rather than have students write their own code to insert, they were provided with an instruction that represented a system call to get root privilege. If they successfully inserted the instruction on the stack and were able to get it to execute, they were rewarded with a “root>>” prompt in the output pane.

The entire buffer overflow lab is accomplished using five web pages with the examples described along with explanatory text on each page. Students in the security class, including non-CS majors, were able to comprehend the tool and successfully complete the lab in a single class session.

5. OUR EXPERIENCE AND PLANS

The initial use of SMS in our computer security course proved highly successful. The basic concepts of buffer overflow, why it is a security threat, and approaches to detect or eliminate them were presented in lecture format, followed by a one-class lab using the SMS tool. Students were able to complete the entire lab in a single class session (50 minutes). Thus, two lessons were devoted to the topic as opposed to three lessons in previous semesters. In addition, non-CS majors who struggled to complete the complexity of the previous hands-on demonstration, were able to successfully complete and understand the web lab version with SMS. Although quantitative analysis of student knowledge gained based on exam questions is difficult due to small sample sizes, the buffer overflow web lab was highly rated on student critiques as an enjoyable and useful experience.

Based on this success, we began investigating the use of the tool in some of our other computer science classes. While the tool was initially developed for the buffer overflow problem, by designing it with a general language/memory architecture, we created an environment that can be easily re-purposed for other applications. For example, in our operating systems class, we have successfully introduced the tool as a refresher on the purpose and operation of stack frames when calling routines. In our computer architecture course, which teaches assembly language programming, we are planning on using the tool to introduce the concepts of stack frames and subroutine calling without the details of the specific language.

Some modifications to the tool are planned to accommodate the different ways of using it. For example, an editor is being incorporated to allow students to create their own high-level programs on the fly based on the simple instruction set and then assemble them to memory. This will reinforce the idea of a Von Neumann architecture in which instructions and data coexist in

memory. It will also allow students to write simple programs that they can “watch” execute. Other courses are also reviewing the tool to determine applicability in their material. Since computer science majors will have seen and used the tool initially in their sophomore year, it will be a familiar environment for other demonstrations.

The tool is public domain and freely available for use. Please contact the authors for further information.

6. REFERENCES

- [1] ACM Curricula Recommendations, <http://www.acm.org/education/curricula-recommendations>, last accessed Sep 11, 2009.
- [2] SPIM, A MIPS32 Simulator, <http://pages.cs.wisc.edu/~larus/spim.html>, last accessed Sep 11, 2009.
- [3] MIPSTER 2.0, <http://www.downcastsystems.com/mipster/default.asp>, last accessed Sep 11, 2009.
- [4] Dale Skrein. CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes. *ACM Journal on Educational Resources in Computing (JERIC)*, Vol. 1, Issue 4, 2001, 46-59.
- [5] Andrew E. Phillips and Jack S. E. Tan. Exploring Security Vulnerabilities by Exploiting Buffer Overflow using the MIPS ISA. *ACM SIGCSE 2003 Bulletin*, Vol. 35, Issue 1, Jan 2003, 172-176.
- [6] Barnett, B. L. A visual simulator for a simple machine and assembly language. *Technical Symposium on Computer Science Education (SIGCSE, Nashville Tennessee, USA)*, 1995, 233-237.
- [7] A Simple Assembly Language Computer Simulator. <http://www.cse.yorku.ca/~peterc/simulator/simulator.html>, last accessed Sep 11, 2009.
- [8] Gerhart, Susan. Animation of Buffer Overflows and Cryptography. *Technical Symposium on Computer Science Education (SIGCSE, Norfolk Virginia, USA)*, 2004.
- [9] Jaishankar Sundararaman and Godmar Back. HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. *Proceedings of the 4th Symposium on Software Visualization (SOFTVIS, Ammersee, Germany)* 2008, 47-56.