A PRESENTATION AND EVALUATION OF GUIDED-LEARNING ACTIVITIES AND PROGRAM VISUALIZATION TOOL, DISSAV, TO TEACH STACK SMASHING

by

Erik Akeyson

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Cyber Security

Charlotte

2022

Approved by:

Dr. Meera Sridhar

Dr. Harini Ramaprasad (Co-Chair)

Dr. Erik Saule

ABSTRACT

ERIK AKEYSON. A presentation and evaluation of guided-learning activities and program visualization tool, dissav, to teach stack smashing. (Under the direction of DR. MEERA SRIDHAR)

The aim of this thesis is to improve student learning of advanced cybersecurity topics, more specifically, stack smashing attacks, by increasing student engagement and interaction. To achieve the aim, this thesis develops a program visualization tool for teaching stack smashing attacks, DISSAV (Dynamic Interactive Stack Smashing Attack Visualization) with an accompanying hands-on activity. DISSAV provides a simulated attack scenario that guides the user through a three-part stack smashing attack. The hands-on activity assists the user throughout conducting an attack while highlighting key stack smashing concepts for students. In addition, this thesis incorporates a collection of *guided-learning* activities into a secure software module. The collection of guided-learning activities help student groups work systematically through increasingly challenging content and questions that eventually help them infer and co-construct knowledge with their peers, unlike traditional lecture-style pedagogy.

This thesis evaluates the effectiveness of DISSAV , the hands-on activity and the guided-learning activities and presents the results of deploying them within a software security module in two sections of an undergraduate, introductory cybersecurity course in the Fall 2021 semester, reaching a total of roughly 100 students. Results from a student survey, including two user interface, six student learning, and six student engagement questions, are presented. Results indicate that the majority of students have positive responses to the student engagement questions while 10.3% of responses are negative. For student learning, nearly 80% of responses are positive while under 4% of responses are negative. This thesis then evaluates data based on a number of factors including age and prior experience with stack smashing attacks,

program visualization tools and C programming to see how effective DISSAV is across different demographics. This thesis finds consistent responses to the student engagement questions across different demographics. Finally, the thesis compares student performance in past semesters where the software security module was taught using traditional approaches, i.e., without using guided-learning activities or program visualization tools, to that in Fall 2021. The thesis concludes that the guided-learning activities and program visualization tool help improve the majority of students' engagement and perceived learning.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. Meera Sridhar for the continuous support throughout my Masters Thesis and research, and for her patience, motivation, enthusiasm, and immense knowledge. Her guidance has greatly helped me through the research and writing of my thesis. I am truly grateful to have her as my advisor and mentor for my thesis study.

I would like to thank my co-advisor, Dr. Harini Ramaprasad, for providing quality groundwork for this topic of research, providing immensely constructive feedback, and helping me present my research ideas more concisely. She has brought significant benefit to my research and writings.

I would also like to thank my committee member, Dr. Erik Saule, for his background knowledge and insightful comments throughout the work of my thesis.

This research was supported by NSF award NSF-DGE # 1947295.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

In 2021, there was a 50% increase in overall cyber attacks per week on corporate networks compared to 2020 [1]. The increasing number of daily cyber threats that companies and governments face results in an increase in the number of security experts desired within these entities. Yet, the global skill shortage in the cybersecurity field is well-known by business owners and experts in the field [2]. Effective cybersecurity education is essential to meet the increasing demand for cybersecurity experts. However, we see that educational institutions within the United States fail to keep up with this growing need for cybersecurity talent [3].

Control-hijacking attacks are a class of cyber attacks that aim to take over a target machine by hijacking the application's flow to achieve remote or arbitrary code execution [4, 5].These types of attacks are quite popular today [6, 7]. A common technique for conducting a control hijacking attack is exploiting a *buffer-overflow* vulnerability [4, 5], a vulnerability that allows an attacker to write data to a buffer that overflows the buffer's capacity, overwriting adjacent memory locations [8]. Buffer overflow vulnerabilities are known to be some of the most dangerous vulnerabilities because they are often used for remote code execution or privilege escalation [9]. Buffer overflow vulnerabilities have the ability to alter video streams from an IP camera [10], eavesdrop on conversations through desktop conferencing IoT gadgets [11], and even start someone's Cosori Smart Air Fryer without their knowledge [12].

A *stack smashing* or *stack-based buffer overflow* attack is a type of buffer overflow attack that targets the call stack in C programs; stack smashing attacks are representative of control hijacking attacks because they both aim to take control over a system. Stack smashing attacks are an important topic to teach and should be considered a

core part of the computer security curriculum at educational institutions due to their impact and consistently high severity rating [13]. However, teaching stack smashing is a complex task as research shows that the C programming language is particularly difficult for novice programmers to understand [14, 15, 16] and vast background information is required. For example, students have to acquire all of the following background in order to grasp stack smashing: (i) parameter passing in C, (ii) how parameters are stored on the stack, (iii) C compilation using `gcc`, (iv) assembly code (to comprehend assembly code instructions on the stack), (v) process memory layout (to understand how the heap, data, and code sections of memory work), (vi) the meaning and usage of `argv` (to grasp how the program passes user input), (vii) buffer storage (to know how character arrays are stored on the stack), (viii) buffer overflow and how the program handles data when unsafe functions, such as `strcpy`, copies a value into a buffer that contains less memory space than the value, (ix) overwriting a return address to comprehend how someone can change the return address of a subroutine, (x) and shellcode to demonstrate the dangers of stack-based buffer overflow attacks [17]. Our goal is to create content that is interactive, engaging, and guided, to help address these teaching and learning challenges.

Program visualization is the process of using graphics to aid in the programming, debugging, and understanding of computer systems [18]. Program visualization aims to expand the types of resources available to teachers and institutions to enhance students' understanding of software topics along with encouraging active engagement. Program visualization tools provide visual representation for the student, with the aim of increasing engagement. Prior work suggests that program visualization is a beneficial resource in the classroom [19, 20, 21, 22, 23, 24]. Researchers develop a number of program visualization tools to educate students on buffer overflow attacks [25, 26, 27, 28]. A number of papers evaluate the overall effectiveness of program visualization tools [29, 30, 31].

Our *guided-learning* activities — which attempt to follow the Process Oriented Guided Inquiry Learning or POGIL [32] style — present models that capture relevant content, allows students to *explore* the models via a series of simple questions, then move on to questions that require them to *infer* concepts based on the models and finally answer questions that require them to *apply* the concepts they infer. The activities encourage students to discover or construct the more complex ideas themselves, compared to being directly taught about it.

In this thesis, we present a program visualization tool, DISSAV: Dynamic, Interactive Stack Smashing Attack Visualization, an accompanying hands-on activity and a collection of guided-learning activities that address the above challenges and facilitate teaching stack smashing. Our program visualization tool teaches students stack smashing attacks through a guided, simulated attack scenario. Our set of guided-learning activities along with DISSAV and the hands-on activity are packaged into a "Secure Software" module. The guided-learning activities include four activities, namely: (1) Introduction to C, (2) Process Memory Layout, (3) Stack Smashing, and (4) Stack Smashing Defenses. The guided-learning activities are designed to be completed by students in small, self-managed groups that test students' knowledge on the current concept before they continue to the next concept. This design encourages students to discover the more complex ideas themselves. We design the following research questions to show our motivation for this work.

1.1 Research Questions

1. Do students find DISSAV and the hands-on activity an engaging resource for learning about stack smashing attacks within the classroom?

2. Do the guided-learning activities improve student learning by encouraging students to discover the more complex ideas themselves?

3. Do DISSAV and the hands-on activity consistently improve student learning and

engagement across all age groups while also effectively teaching stack smashing even to students with no prior experience on the topic?

To answer the above research questions, we deployed the guided-learning activities, DISSAV, and the hands-on activity within two sections of a junior level undergraduate course in the Fall 2021 semester, reaching a total of roughly 100 students. We also administered a student survey to obtain student feedback on DISSAV and on their perceived learning and engagement with the tool and activity. The student survey consists of two user interface questions, six student learning questions, and six student student engagement questions. This thesis reports the averages of responses from 26 students (who consented to have their responses analyzed) for all questions from our student survey to find that 73% of students provided positive responses, 21% provided neutral responses, and 6% provided negative responses. We find that over 75% of students provided positive responses to the student learning questions. We also find that over 60% of students provided positive responses to the student engagement questions. We compare grade averages from prior semesters to the Fall 2021 semester using a subset of questions from the secure software quiz and an exam. We find statistically significant student learning improvements when analyzing the averages of a subset of quiz questions but also finds no statistically significant improvements in averages from the exam grades.

1.2 Main Contributions

1. We design, develop, and deploy a program visualization tool, DISSAV, and accompanying hands-on activity that teaches stack smashing attacks to undergraduate students through a guided, simulated attack scenario.

2. We deploy four guided-learning activities that cover an introduction to C, process memory layout for C programs, stack smashing attacks, and defenses to stack smashing attacks.

3. We evaluate the collection of guide-learning activities, the program visualization tool and the hands-on activity through a student survey, demographic analysis of the student survey, and comparative grade analysis with previous semesters.

1.3 Publications

Parts of this thesis have been published in another venue. The Colloquium for Information Systems Security Education (CISSE) published "Dissav: A dynamic, interactive stack- smashing attack visualization tool," in vol. 9, no. 1, in 2022.

1.4 Roadmap

Chapter 2 provides background information about stack smashing attacks. Chapter 3 describes the design of DISSAV. Chapter 4 describes the collection of guided-learning activities previously created. Chapter 5 describes the deployment and analysis of the student survey, DISSAV, the hands-on activity, and the guided-learning activities. Chapter 6 discusses related work. Chapter 7 presents our conclusion and future work.

CHAPTER 2: BACKGROUND

2.1 Stack Smashing Attacks

In C programs, a *call stack*, also referred to as an *execution stack*, is a data structure that holds information on active functions of a program [33]. A stack frame is pushed onto the call stack when a function is called and is popped once the function execution has completed. Each stack frame contains a return address to direct program execution back to the calling function after the running function completes execution. In C programs, execution starts with the `main` function and `main`'s stack frame is the first to be pushed onto the call stack. The `main` function accepts an arbitrary number of parameters provided by the user through an array called `argv`, which goes into `main`'s stack frame.

In a stack smashing attack, the attacker attempts to corrupt the call stack [17] by overwriting the return address of a stack frame to point to a place in memory where the attacker stores their malicious code of choice [17]. The attacker does this by locating and exploiting a buffer overflow vulnerability in code written using unsafe functions, e.g., `strpcy`, to copy more data into a local buffer than it can hold. If the value being copied into a buffer takes up more space than the buffer can hold, the program stores the data in adjacent memory. It is possible for an attacker to overwrite the return address in this process because the program stores local variables at a lower memory address than the return address. By cleverly overwriting the local buffer (which goes on to the call stack as part of the running function's stack frame) with code input through `argv`, the attacker overwrites the return address of the stack frame.

For the *payload* (malicious input) construction, the attacker uses three main components: (1) the *NOP sled*, (2) the *shellcode* (the attacker-chosen malicious code),

and (3) a repeated malicious return address (the address of the shellcode). Each of these components are described in more detail below:

- **the NOP sled**: The payload starts with a series of `nop`, or "no operation" assembly language instructions, called a NOP sled. A NOP instruction performs a null operation that simply continues execution and is usually used to delay execution for purposes of timing [17]. The attacker wants their new return address to point to the beginning of the shellcode, which executes the shellcode. The issue is the attacker needs to know the exact address where the shellcode begins in memory. It is very difficult to calculate the correct return address due to stack randomization and other runtime differences [34]. An attacker can estimate where the shellcode begins in memory by guessing the offset of the shellcode from the beginning of the stack, however, this is not an efficient process and would take at best a hundred tries, and at worst a couple of thousand [17]. To account for this, the attacker places a long series of NOP instructions in memory. Once program execution lands in the NOP sled, program execution "slides" to the beginning of the shellcode and begins execution of the shellcode. Landing in the NOP sled ensures complete shellcode execution. The shellcode will most likely crash or result in a segmentation fault if the program returns to an address anywhere but the beginning of the shellcode.

- **the shellcode**: The program the attacker wishes to execute is often referred to as shellcode because it starts a remote shell on a machine. The program stores the shellcode in the local variables section of its corresponding stack frame since the program stores the payload in a local buffer.

- **repeated malicious return address**: The last component of the payload is the new return address (the address of the payload), which is repeated several times. Since the exact position of the return address on the stack is also difficult

to calculate, because its value changes each time the program compiles, the attacker repeats the new return address in the payload to increase the chances the new return address is correctly positioned on the stack [35].

The attacker then passes the payload as a parameter to the program and the program stores the payload in `argv`. The program stores `argv` as a parameter to `main` in its stack frame. The `strcpy` function then copies the payload contained in `argv` into a local variable buffer. The program returns to the malicious return address if a correct payload is used. The program executes the shellcode once program execution has reached the malicious return address.

Although stack smashing attacks only affect languages with unsafe functions, they have widespread impact due to the large amount of legacy code used in today's applications [36].

2.2 Program Visualization

Program visualization is the process of using graphics to aid in the programming, debugging, and understanding of computer systems [18], which educators and developers most commonly implement as web interfaces. Program visualization aims to expand the types of resources available to teachers and institutions to enhance students' understanding of software topics along with encouraging active engagement. Teachers use such systems in lectures to illustrate the changes in program states during the execution of programs [37]. These tools also allow students to independently practice topics that they find difficult [37]. Educators and researchers develop a number of program visualization tools to assist students in the classroom [19, 20, 21, 22]. Program visualization tools cover a wide variety of topics from simple introductory algorithm comprehension [38, 21] to more specialized areas such as call stack visualizations [28]. Prior work shows that program visualization is a beneficial resource in the classroom [29, 26, 39, 24].

CHAPTER 3: DISSAV: DYNAMIC, INTERACTIVE STACK SMASHING ATTACK VISUALIZATION

DISSAV is an interactive program visualization tool that aims to teach stack smashing attacks to undergraduate students [40][1]. Our overarching goals are to engage a broader and more diverse student body and foster student interest in the field of cybersecurity and ultimately improve student learning outcomes in cybersecurity topics. We aim to achieve these goals by teaching important cybersecurity concepts such as stack smashing attacks in an interactive and engaging manner.

DISSAV allows the user to construct a customizable stack smashing attack scenario, guided through incremental steps, to promote engagement and understanding. The user can change the program and payload through dynamic input while working with the tool. First, the user creates up to three functions and adds them to a program named `intro.c`. Next, the user can optionally construct a payload to provide as input to the program. Lastly, the user executes the program to interact with the call stack visualization and to complete a successful stack smashing attack.

3.1 DISSAV workflow

3.1.1 Create the Program

In this phase, the user incrementally builds a program named `intro.c` by creating one or more functions and adding them to the program. Our `Create a function` phase allows the user to create a basic function by providing a function name and optionally adding local variables and parameters, as shown in Figure 3.1. The user can create a local variable or parameter by specifying the name, selecting a data type

[1]This chapter includes previously published ([40]) joint work with Erik Akeyson, Meera Sridhar, and Harini Ramaprasad

from a dropdown box, and declaring a value. DISSAV currently supports `char`, `int`, and `char[]` data types.

Figure 3.1: Function name, parameters, and local variables

Additionally, when creating a function, the user can 1) add a call to an unsafe C function; 2) pass `argv[1]` as a parameter; and 3) call another function that has been previously added to the program. The first two of these features play key roles in the stack smashing attack and the ability to call an additional function enhances the call stack visualization. As code is added to the function being created, DISSAV displays the code to the left of the buttons shown in Figure 3.2.

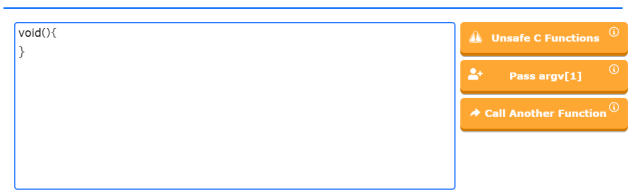

Figure 3.2: Function Display

After a function is created, a colored pointer directs the user to add it to the program, `intro.c`, and DISSAV displays the program on the right side of the screen, as shown in Figure 3.3. DISSAV dynamically updates the program code as the user adds new (currently up to three) functions.

Our design supports the minimal functionality needed to create a C program that can be used to construct a stack smashing attack and allows users with even the most basic understanding of programming to build valid C programs. Our design allows the user to view the program code, `main` function, the role of `argv` and function calls from the `main` function, all while constructing the program.

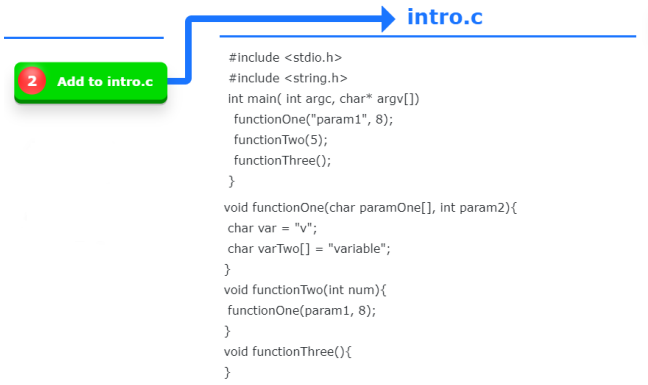

Figure 3.3: Program Display

3.1.2 Construct the Payload

After creating the program, the user can choose to use the `Construct Payload` phase to create a custom payload, by clicking a checkbox indicating that they want to attempt a stack smashing attack. If the user chooses not to construct a payload, DISSAV allows them to provide simple strings such as “cat" or integers such as 15 as input to the program instead.

If the user chooses to construct a Payload, DISSAV displays a dynamic diagram that represents each part of the payload in a separate color, as shown in Figure 3.4. As the user continues through each part of the payload, DISSAV highlights the corresponding colored section with a border.

Figure 3.4: Dynamic payload diagram

Our payload consists of three parts. Each part contains hints on how to construct the corresponding section, as shown in Figure 3.5. The user begins with creating a NOP sled, then adds the shellcode and finally ends with a repeating return address

as explained in section §2. We implement this design to provide sectioning of the

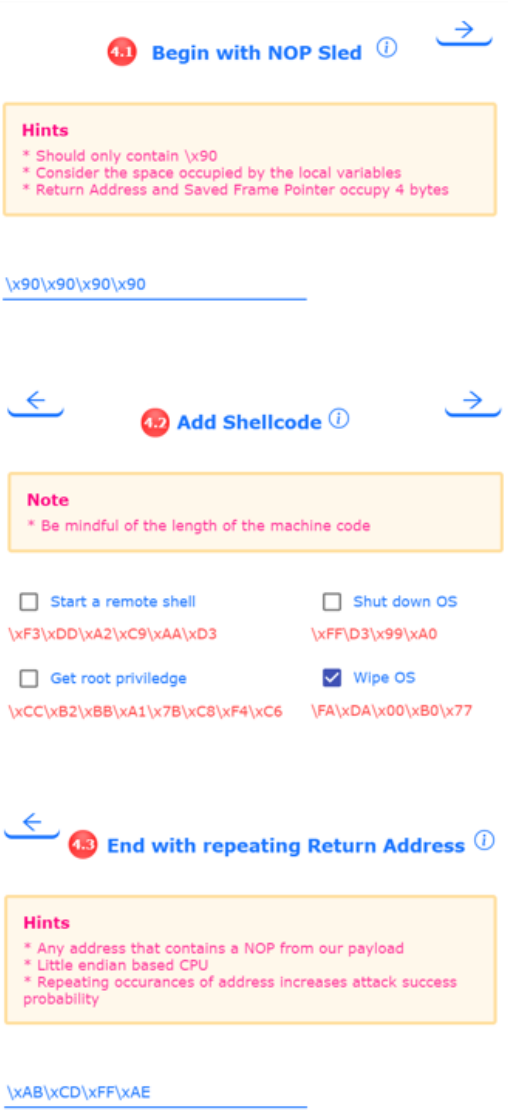

Figure 3.5: Construct Payload

payload, which allows the user to analyze and work on individual pieces to break down each concept.

3.1.3 Execute the Program

After completing the `Create the Program` phase and optionally the `Construct Payload` phase, the user moves to the `Execute the Program` phase. The user clicks the `Start` button shown in Figure 3.6 to start program execution. Once program

execution starts, DISSAV passes `argv` to the `main` function, where `argv` is either the constructed payload or a simple string that the user provides as input.

Figure 3.6: Start Button

The user clicks the `Next` button shown in Figure 3.7 to step through the program. DISSAV pushes / pops a function each time the user clicks the `Next` button and passes the user's input to functions that take `argv` as a parameter (either directly or copied into local variables). Once the program reaches the end of `main`, DISSAV

Figure 3.7: Next Button

displays the `Finish` button shown in Figure 3.8, which pops the `main` function and ends program execution.

Figure 3.8: Finish Button

DISSAV provides dynamic visual representations for the call stack, stack frame, and program code during program execution. We discuss the details of each component next.

3.1.3.1 Visualize Call Stack

A key aspect of DISSAV is the `Call Stack`, which displays the current state of the call stack during program execution, as shown in Figure 3.9. DISSAV pushes / pops stack frames onto the `Call Stack` as the user steps through each function call. For each function that is currently on the `Call Stack`, DISSAV displays a box with

the name of the function at the center and provides a dropdown button that can be opened to view the details of the function's stack frame (We explain this component in Section 3.1.3.3). DISSAV uses a red background color for unsafe functions and does not provide stack frame details for the unsafe (library) functions themselves since those are not created by the user.

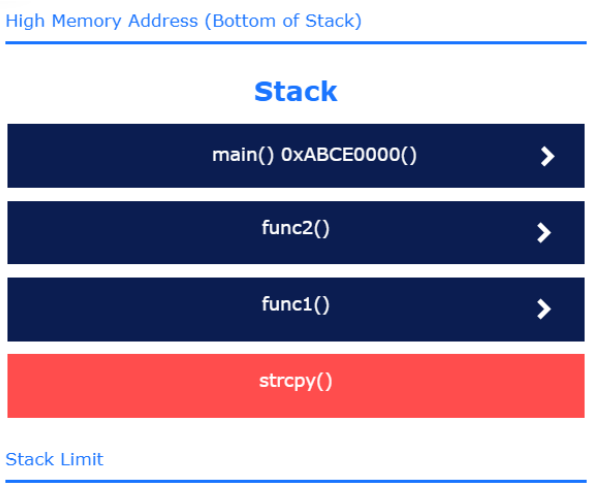

Figure 3.9: Call Stack

We do not intend for DISSAV's `Call Stack` to be a detailed program execution call stack similar to ones presented in Jeliot [21], Jype [20], and ViLLE [23], which include details such as visualization of the control flow and object structures and a visualization for each line of code in the program. We design DISSAV as an interactive call stack visualization tool that only provides information relevant to a stack smashing attack. We choose this design to provide a simple, dynamic view that is easy for the user to comprehend. We implement the dropdown functionality for each stack frame to maintain a cleaner look and avoid overwhelming the user with all the details at the same time. DISSAV allows the user to return to the `Create the Program` phase at any time, to make changes to their functions and see how the changes impact the `Call Stack`.

3.1.3.2 Visualize Program Code

DISSAV highlights the corresponding program line for each movement of a stack frame, as shown in Figure 3.10. DISSAV highlights the function's name and parameters when the function is pushed onto the stack and highlights only the name of the function when popping the function off the stack.

```
void funcOne(char p[]){
 char v[] = "v";
 strcpy(v, p);
}
```

Figure 3.10: Calling `strcpy`

The parameter `argv` plays an essential role in stack smashing attacks. DISSAV uses a dark blue font color to represent the `argv` parameter, as shown in Figure 3.11. DISSAV shows `argv` starting as a parameter in the `main` function, moving as a parameter to a function called from the `main` function, then finally being passed to `strcpy`. The different font colors and highlights help the user make a connection between the program execution, the movement of the stack and the movement of `argv`.

```
int main(int argc, char* argv[])          funcTwo(argv[1]);
 void funcTwo(char userInput[])          strcpy(v,userInput);
```

Figure 3.11: Following `argv`

3.1.3.3 View Stack Frame

DISSAV provides a detailed stack frame display, which contains the parameters, return address, saved frame pointer, and local variables, all with their corresponding memory addresses, for each stack frame that is open (i.e., for which the user clicks on the dropdown button), as shown in Figure 3.12. DISSAV displays a label next to each section of the stack frame (e.g. Parameters), to describe the data within the section. DISSAV updates the stack frame dynamically if the user passes input to the corresponding function. We choose this design to provide a simple representation of

the stack frame that is easy to understand and track data in. The view assists the user in understanding how data is pushed and moved within the stack frame.

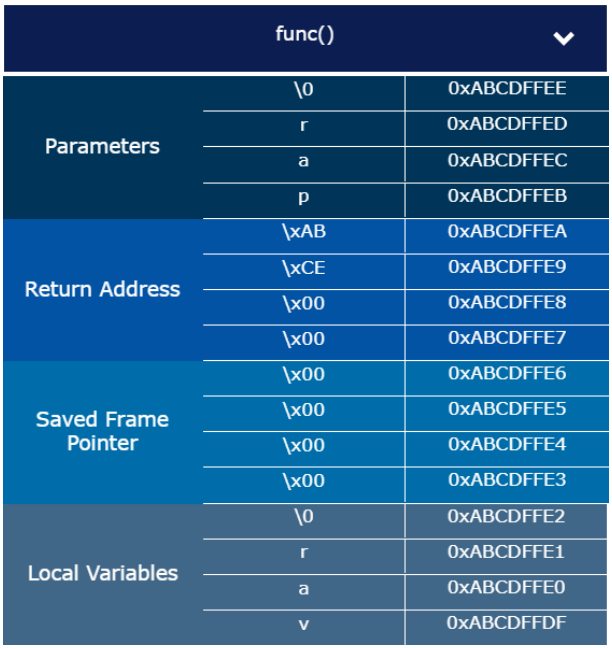

Figure 3.12: Stack Frame

3.1.3.4 Complete a Stack Smashing Attack

DISSAV allows the user to attempt to complete a stack smashing attack. The user does so by creating a function that contains a buffer overflow vulnerability, constructing a payload that attempts to exploit the vulnerability, and then executing the program with the payload. An attack is successful if a correct payload is constructed. The user's goal is to overwrite the return address to an address that falls within the NOP sled of the payload. The stack frame display assists the user in choosing a correct return address and calculating the length of the payload. The set of correct return addresses varies based on the current state of the call stack, the parameters, and local variables. DISSAV tracks all functions where a successful attack has taken place and displays them along with an attack status for feedback, as shown in Figure 3.13.

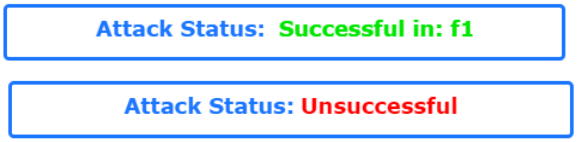

Figure 3.13: Attack Status

3.2 DISSAV highlights and limitations

3.2.1 Engagement in Program Visualization

The early 2000s saw a great interest in the research of engaging the learner in an active way with software visualization tools. Many influential papers [41] define six categories of engagement: *No viewing*, *viewing*, *responding*, *changing*, *constructing*, and *presenting*. DISSAV provides engagement in the *constructing* category, allowing the user to not only provide dynamic input, but to construct and then see a visual representation of their own code. Researchers have found *constructing* to be more engaging than *changing* [29]. We aim to implement *responding* and *presenting* in future work to increase student engagement.

3.2.2 Ease of Use

DISSAV is an interactive web-based application built using React JS for the user interface or front-end. It is easily accessible via a weblink and has been tested on the most commonly used browsers, Chrome, Safari and Firefox. It requires no prior knowledge of C and minimal programming experience. DISSAV brings the user to a simple landing page (shown in Figure 3.14) where they are able to click on the `Begin` button. The user is guided through the DISSAV workflow by the numbered markers shown in Figure 3.15. Most of the markers are simply buttons that the user clicks to go to the next stage and require no inference. Markers one (`Create a function`), and four (`Construct Payload`) require the user to infer some knowledge. The user can always return to the first section for code modifications.

Figure 3.14: Landing Page

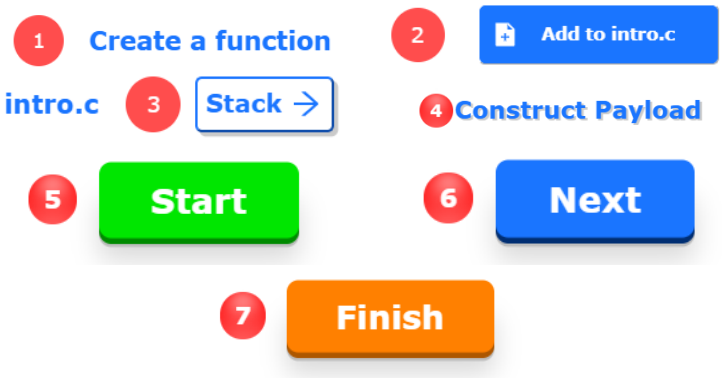

Figure 3.15: Instructional Steps

3.2.3 Limitations

DISSAV supports a limited version of a C program that only features representative aspects to allow a simple stack smashing attack. A function may only contain parameters, local variables, a single `strcpy` function call, and calls to other functions within the program; no other program statements are supported. Parameter and local variable data types are limited to `char`, `int`, and `char[]`. During program execution, the `Next` button, the `Call Stack` and program highlights correlate to each function call and not to each line of code. DISSAV provides a limited implementation of program execution that features only representative aspects. The C code that DISSAV displays has no connection to the C programming language, simply a representation. The next button, call stack, and highlighted code correlate to each function call, not each line of code which is common in step through program execution. Finally, since DISSAV is a web-based application, only users with access to a computer with internet connection can use DISSAV.

3.3 DISSAV Accompanying Hands-on Activity

We design and deploy a hands-on activity to accompany DISSAV. The activity guides the student through a stack smashing attack while touching on concepts covered earlier in the secure software module for review. The activity includes five multiple choice, four short answer, three screenshot upload, and one matching question. The activity starts by covering simple C programming concepts (e.g., data types) then continues to the three phases discussed in Section 3.1. The activity provides instructions on creating a vulnerable function, constructing a payload, and executing the function. The activity encourages students to use "different strings of different lengths and number of words" before attempting to construct an attack payload. We incorporate this feature to allow students to test different inputs and to experiment and visualize how the computer passes and stores data on the stack before construct-

ing a full payload. While the activity provides instructions for payload construction along with hints, the exact process is not given. Students must experiment by using different numbers of NOP sleds, retrieving a good malicious return address, and formatting the return address. We encourage students to learn how the return address is overwritten and how the shellcode is executed through trial and error, similar to a real stack smashing attack. We include questions that cover major variables on the call stack (e.g., `argv` and the character '\x') to highlight their importance. At the end of the activity we provide more high-level questions — e.g., how did they determine their new return address, how did they determine the length of the payload, etc. — with the aim of emphasizing key concepts in a stack smashing attack. The aim of the activity is to have students build up to these more abstract concepts such as how the computer passes data from `argv` to `main`'s stack frame and the execution of shellcode on the call stack to adequately understand stack smashing attacks.

CHAPTER 4: GUIDED-LEARNING ACTIVITIES FOR TEACHING STACK SMASHING ATTACKS

In this chapter[1], we briefly discuss a sequence of guided-learning activities developed to teach concepts that relate to stack smashing attacks. The activities attempt to follow the Process Oriented Guided Inquiry Learning or POGIL [32] style for creating engaging activities. POGIL-style activities present learning models (a model could be program code, graphical representations of concepts, tables of data, etc., depending on the topic). Students first start by *exploring* the models via a series of simple questions, then move on to questions that require them to *infer* concepts based on the models and finally answer questions that require them to *apply* the concepts they infer. Such activities encourage students to discover or construct the more complex ideas themselves, compared to being directly taught about it and have been shown to improve student mastery of content in a variety of areas and topics [42].

The four guided-learning activities are *Introduction to C*, *Process Memory Layout*, *Stack Smashing*, and *Defenses*. The activities start with basic concepts of C programming and then gradually move to more complex ideas such as a stack smashing attack payload. The end of each activity provides external resources for students to review and a discussion question to encourage further learning among each group. Two out of the four guided-learning activities have been posted as "Activities for Review" — which is the first stage in the POGIL activity endorsement process — in the POGIL Activity Clearinghouse or PAC [43] . The remaining two activities are being prepared for submission to the PAC in the near future. We hope to have

[1]This chapter includes previously unpublished work by Yates Snyder, Meera Sridhar, and Harini Ramaprasad.

all four activities classroom-tested and eventually endorsed by POGIL. To the best of our knowledge, there are currently no POGIL-endorsed activities for advanced cybersecurity topics [44].

4.1 Introduction to C

The first activity introduces the C programming language to students and contains 32 questions. The goal is to teach students how to create and run a C program that uses command-line arguments. The first learning objective is to teach students how to create simple C programs using variables, functions, and arrays. The second objective is to have students understand the structure of strings in C as referred to in Figure 4.1, their memory composition in bytes, and the unsafe C function `strcpy()`.

smallString

s	m	a	l	l	\0
0x5556	0x5557	0x5558	0x5559	0x555A	0x555B

Figure 4.1: Structure of strings in C

4.2 Process Memory Layout

The next activity covers concepts relating to how a computer handles and processes data in memory and contains 38 questions. The goal of this activity is to have students understand the process memory allocation details required to conduct a stack smashing attack. This activity discusses what a stack pointer is, what a program counter is, what constitutes a stack frame, how and when they are added and removed from the call stack, and how to recognize the current state of the call stack when given a program at the current place of execution. The objective is to have students be able to describe the different segments within the main memory of a computer, their growth directions, and relative positions as referred to in Figure 4.2. Figure 4.2 shows the layout of the call stack and it's state at a given point of execution. The arrow pointing to the `strcpy` function represents the current point of execution. We also

cover identifying the values of important variables, such as `argv` and `argc`, when given a command-line input.

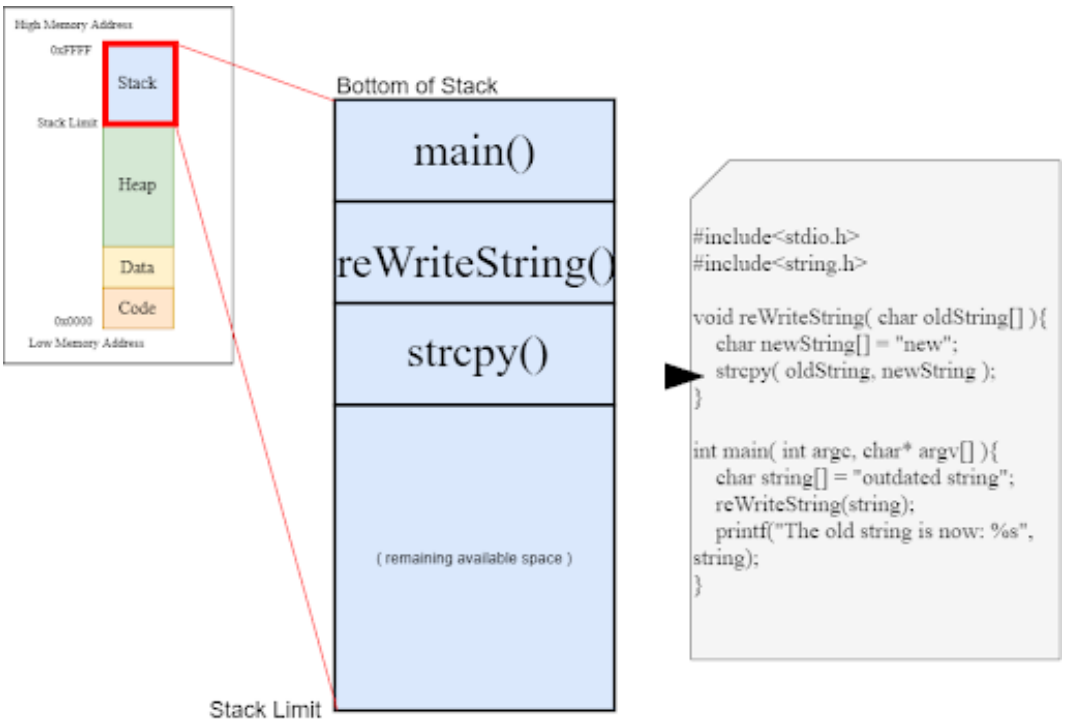

Figure 4.2: Relative positions of memory

4.3 Stack Smashing

After the students understand basic C programming concepts and process memory layout, we move to the stack smashing attack activity that contains 24 questions. The goal of this activity is to have students understand how previously taught concepts can be used to complete a stack smashing attack. The learning objective is to have students be able to describe the relevance and importance of the return address and NOP sleds in a stack smashing attack as shown in Figure 4.3. Figure 4.3 displays the three components of the payload that we describe in Section 2.1. We represent the NOP sled with a yellow background, the shellcode with a green background, and the attacker's new return address with a red background. We use an arrow that points from the payload figure on the lower left to `argv` to represent that the program uses `argv` to pass the payload to the function. We discuss the contents of memory at

different stages of program execution when given an example of a stack smashing attack. Lastly, we cover how to determine the size of memory addresses, the size of the payload required, and the size of each of the payload's components when given a diagram of a stack.

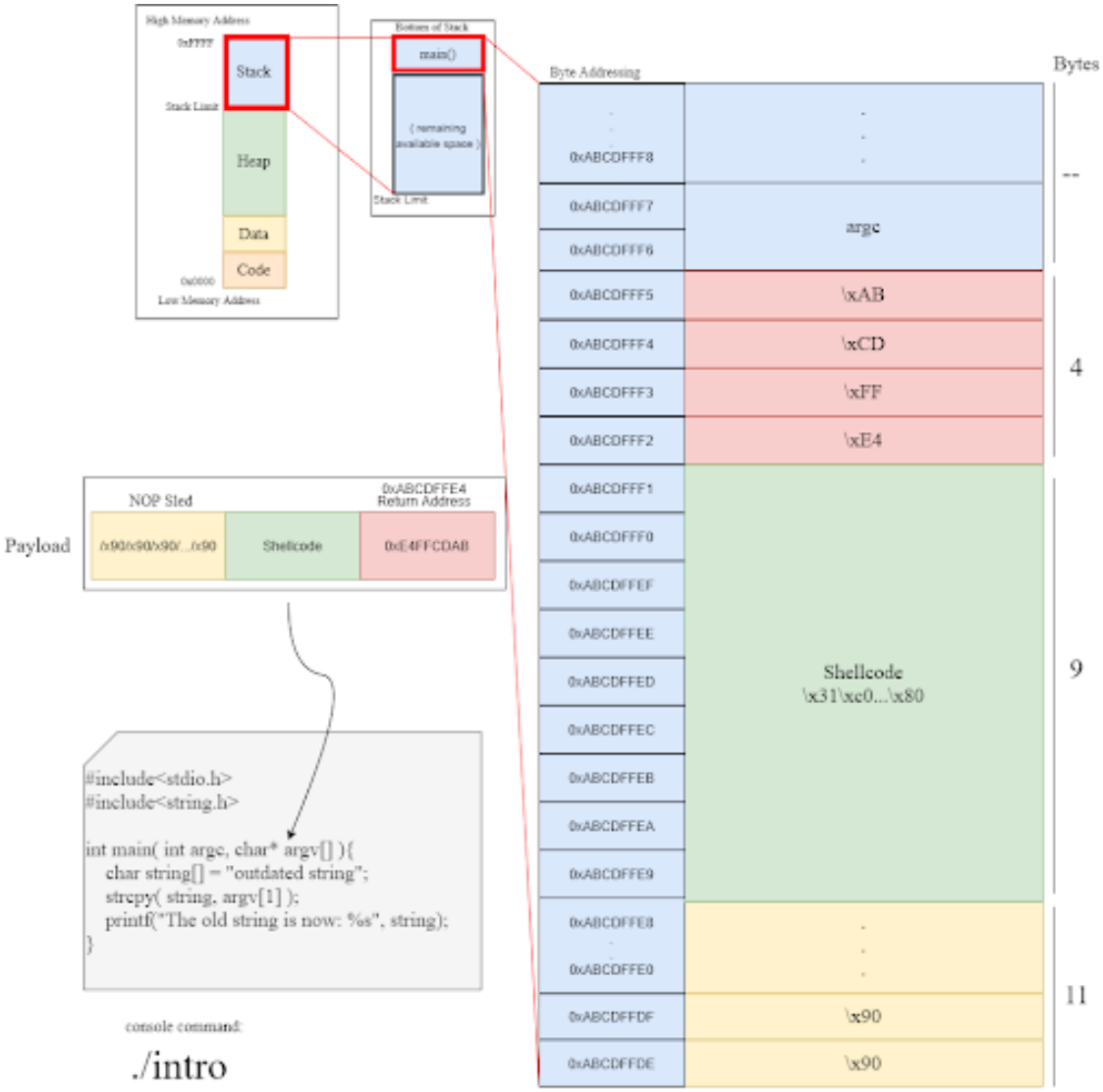

Figure 4.3: Stack Smashing

4.4 Stack Smashing Defenses

After students understand and are able to complete a stack smashing attack, we cover the defenses activity that contains 16 questions. The goal of this activity is to present a number of techniques to prevent stack smashing attacks. The learning ob-

jective is to have students be able to explain how and why defenses such as ASLR [45], non-executable stacks [46], and stack canaries [47] can help prevent a stack smashing attack. A stack canary is a known value that the system places between the buffer and important values such as the return address that tells the system if the data within the stack frame has been overwritten. Using Figure 4.4, we ask students to select an acceptable location to place the stack canary. We also discuss why `strncpy()`is safer than `strcpy()`.

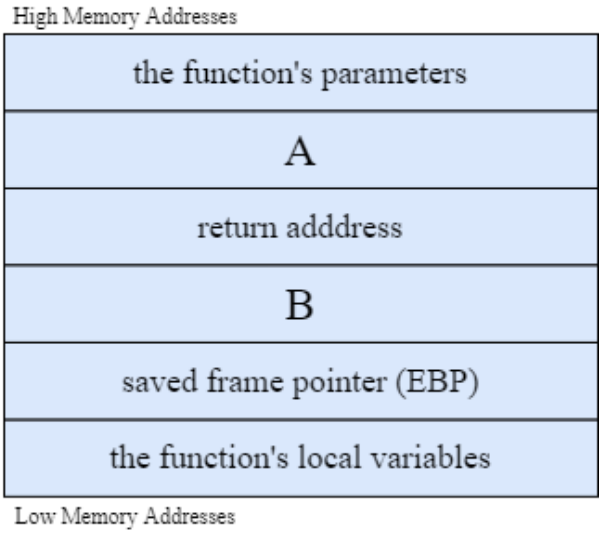

Figure 4.4: Stack Canary

CHAPTER 5: DEPLOYMENT AND ANALYSIS OF THE STUDENT SURVEY, DISSAV, THE HANDS-ON ACTIVITY, AND THE GUIDED-LEARNING ACTIVITIES

5.1 Deployment of Resources

We deploy the guided-learning activities, the program visualization tool, DISSAV, and the hands-on activity within the secure software module of two sections of an undergraduate, introductory cybersecurity course at UNC Charlotte in the Fall 2021 semester, reaching a total of roughly 100 students. The module includes a set-up activity to help students create the correct environment to run C code. We first release the activities discussed in Chapter 4, followed by DISSAV and the hands-on activity described in described in Chapter 3 and then finally the student survey presented in Chapter 5.2. It is worth noting that we deploy the activity, *Introduction to C*, in two parts, as completing the activity within in a single class session is extremely difficult for students.

5.2 Student survey on DISSAV and the hands-on activity

We design and deploy a student survey (approved by UNC Charlotte 's Institutional Review Board or IRB) to obtain feedback on DISSAV and the accompanying hands-on activity. The survey asks two user interface questions, six student learning questions, and six student student engagement questions. The survey also includes demographic questions such as age, gender, and experience level in three different areas. We summarize the responses from the non-demographic questions next and analyze the demographic data in Section 5.3. Throughout the survey and demographic analysis, we use the term *positive* to refer to the responses of Strongly Agree or Agree, *neutral*

to refer to Neither Agree Nor Disagree, and *negative* to refer to Disagree or Strongly Disagree.

We summarize the responses from 26 students who consented to the student survey in Table 5.1. We provide overall averages first and then discuss the student learning and engagement question responses in more detail. Overall, 73% of students provided positive responses, implying that they find DISSAV and the hands-on activity to be good resources in the classroom. 21% of students provided neutral responses, implying that they see no benefit nor drawback to the resources. Only 6% provided negative responses, implying that they find DISSAV and the hands-on activity not to be beneficial. Additionally, we see that only one question falls below the 60% mark for Strongly Agree or Agree, namely "I was so involved in the activity that I lost track of time."

For a more detailed view on student engagement and learning, we analyze the respective sections of the student survey. In the Student Learning section, we see that 74.4% of responses are positive, implying that DISSAV and the hands-on activity improve their perceived learning. An average of 18% of responses are neutral, implying that they find the resources did not make a difference in their learning, while only 7.6% of responses are negative, implying that they find the resources not beneficial for learning. For Student Engagement, 58.3% of responses are positive, implying that they find DISSAV and the hands-on activity engaging. We see that 26.3% of responses are neutral, implying that the resources neither help nor hurt their engagement. We see that 15.4% of responses are negative, implying that they find the resources not to be engaging. In summary, a majority of the students find the activity useful and agreed that it assisted with their engagement, while nearly 75% of students agreed that it assisted with their learning. It is worth noting that one student answered "Strongly Disagree" to the last question about whether they prefer learning with this style of activity or not. As they did not prefer this style of learning, the student might

Table 5.1: Student Survey questions and responses

Question (N=26)	Strongly Agree	Agree	Neither A / D	Disagree	Strongly Disagree
User Interface					
The application design is attractive (graphics, interface, layout)	7 (27%)	14 (54%)	5 (19%)	0 (0%)	0 (0%)
The text font (size and style) and colors are clear and consistent.	8 (31%)	15 (58%)	3 (11%)	0 (0%)	0 (0%)
Student learning					
The learning content and/or previous activities were sufficient to help me understand relevant concepts and do the activity smoothly.	5 (19%)	12 (46%)	6 (23%)	2 (8%)	1 (4%)
The content and structure of the activity helped me gain confidence in the concepts.	7 (27%)	10 (38%)	9 (35%)	0 (0%)	0 (0%)
The contents of the activity are relevant to my interests.	5 (19%)	13 (50%)	7 (27%)	1 (4%)	0 (0%)
It is clear to me how the contents of the activity are related to the targeted concepts.	9 (34%)	16 (61%)	1 (4%)	0 (0%)	0 (0%)
The activity helped me reinforce relevant concepts.	9 (34%)	16 (61%)	1 (4%)	0 (0%)	0 (0%)
This activity is an adequate teaching method for the included concepts.	8 (31%)	13 (50%)	3 (11%)	0 (0%)	2 (7%)
Student Engagement					
I found the activity to be fun/highly engaging (i.e., it does not become monotonous or boring).	7 (27%)	10 (38%)	8 (30%)	1 (4%)	0 (0%)
Completing the individual tasks/phases of the activity gave me a satisfying feeling of accomplishment.	10 (39%)	7 (27%)	7 (27%)	2 (8%)	0 (0%)
I was so involved in the activity that I lost track of time.	0 (0%)	6 (21%)	10 (38%)	7 (27%)	3 (6%)
This activity is appropriately challenging for me.	7 (27%)	12 (46%)	6 (23%)	1 (4%)	0 (0%)
I would recommend this activity to others.	9 (34%)	11 (42%)	5 (19%)	1 (4%)	0 (0%)
I prefer learning with this style of activity to other styles that I have experienced.	7 (27%)	13 (50%)	5 (19%)	0 (0%)	1 (4%)

have had a negative overall experience with DISSAV , leading to negative responses. In the future, we plan to add more questions relating to the type / style of learning students prefer in general.

For more open-ended responses, we ask two short answer questions about DISSAV and the hands-on activity. First, we ask students to "list two strong aspects of the activity". We find three aspects of the tool that are common among several student answers. The most common aspect that students like is the visual representation of different components. Another common, strong aspect is the ease of navigation throughout the tool. Finally, students also state that they find the tool engaging. Then, we ask students to "give two suggestions to improve the activity" and find two suggestions that are common among several student answers. The most common suggestion is to provide more explanation or hints. While explanations / hints are provided in the activity, some students feel that more detail would make the activity smoother. The other common suggestion is that the tool needs User Interface improvements, specifically better window scaling for different laptop and screen sizes.

5.3 Demographic Analysis of Student Survey

To evaluate the effectiveness of DISSAV and the hands-on activity across different demographics, we collect and analyze information on age, gender, and level of prior knowledge in three different areas. The demographic analysis utilizes the responses from the 26 students who consented to the student survey. We do not analyze gender data because we found the sample sizes to be too small for effective statistical analysis. We separate each demographic group based on prior experience with stack smashing, program visualization tools and C programming into groups with no experience, little experience, and some experience.[1] The percentages we use in the tables

[1]We now recognize that the phrases "some experience" and "little experience" could be understood as similar experience levels. In future work, we plan to provide improved phrasing so that students can easily distinguish between the terms. We emphasize the responses from students with no experience within each applicable demographic group (i.e., demographic groups based on prior experience) as the large majority of student within the class have little or no previous experience

showing demographic analysis results represent a portion of the total responses for each demographic, not a portion of the student sample size. For example, if there are six student learning questions answered by 26 students, then there are 156 responses. If 40 of the responses are "Agree", then the percentage for "Agree" would be 25.6%. Our goal is for DISSAV and the hands-on activity to consistently improve student learning and engagement across all age groups while also effectively teaching stack smashing even to students with no prior experience on the topic.

We first analyze whether responses to student learning questions are consistent across different age groups. We summarize the results in Table 5.2[2]. We see that nearly 80% of responses to the student learning questions from students between the ages of 18 and 22 are positive, showing that DISSAV and the hands-on activity improve their perceived learning. Less than 1% of responses from students within this age group are negative, indicating that DISSAV and the hands-on activity does not improve their perceived learning. We emphasize this age group because it is the most common age group for undergraduate students. The data shows that DISSAV and the hands-on activity consistently improve the majority of students' perceived learning in all groups.

We analyze whether responses to student engagement questions are consistent across different age groups. We summarize the results in Table 5.3 We see that over 54% of responses from students in all groups are positive. We see that 9% or less of responses from students under the age of 25 are negative. We discuss the responses from students under the age of 25 as they make up 88% of the sample size. While we do see an increase in negative responses in students that are above 26 compared to other age groups, the sample size is extremely small which causes even a single response to weigh heavily on the overall percentages. Overall, the data shows

with these topics.

[2]The two age groups, 18-22 and 22-25 have an overlapping age of 22. We will address this issue in future surveys.

that a majority of students in all age groups find DISSAV and the hands-on activity engaging.

Table 5.2: Effect of age on student learning

Age	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
18 - 22	14.9%	64%	19.3%	<1%	<1%	19
22 - 25	50%	20.8%	20.8%	8.4%	0%	4
26+	77.8%	11.1%	0%	0%	11.1%	3

Table 5.3: Effect of age on student engagement

Age	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
18 - 22	21%	44.7%	27.2%	5.2%	1.9%	19
22 - 25	29.1%	25%	37.5%	4.2%	4.2%	4
26+	50%	11.1%	5.5%	22.3%	11.1%	3

We analyze whether responses to student learning questions are consistent across groups with different prior knowledge of stack smashing. We summarize the results in Table 5.4. We see that over 77% of responses from all groups are positive, indicating that DISSAV and the hands-on activity improve their perceived learning. We see that over 78% of responses from students with no experience with stack smashing attacks are positive while only 5.5% of responses within this group are negative. Overall, our data shows that DISSAV and the hands-on activity have a positive impact on the large majority of students' perceived learning.

We analyze whether responses to student engagement questions are consistent across groups with different prior knowledge of stack smashing. We summarize the results in Table 5.5. We see that over 60% of responses in each group are positive while students with some experience find DISSAV and the hands-on activity highly engaging as 75% of responses from students within this group are positive. The data

shows that the majority of students in all groups consistently find DISSAV and the hands-on activity engaging.

Table 5.4: Effect of prior stack smashing knowledge on student learning

Experience Level	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
No experience	27.8%	51.1%	15.6%	2.2%	3.3%	15
Little Experience	25.9%	51.8%	20.3%	2%	0%	9
Some experience	33.3%	50%	16.7%	0%	0%	2

Table 5.5: Effect of prior stack smashing knowledge effect on student engagement

Experience Level	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
No experience	24.4%	36.7%	26.6%	10%	2.3%	15
Little Experience	24%	40.7%	27.8%	2%	5.5%	9
Some experience	41.7%	33.3%	16.7%	8.3%	0%	2

We analyze whether responses to student learning questions are consistent across groups with different experience levels with program visualization tools. We summarize the results in Table 5.6. We see that over 74% of responses from students in all groups are positive while over 83% of responses from students with some experience are positive. We see that less than 6% of responses from each group are negative. We see that over 83% of students with no experience with program visualization tools provided positive responses showing that DISSAV and the hands-on activity improve their perceived learning. Overall, our data shows that DISSAV and the hands-on activity have a positive impact on the large majority of students' perceived learning regardless of prior experience with program visualization tools.

We analyze whether responses to student engagement questions are consistent across groups with varying levels of experience with program visualization tools. We summarize the effects in Table 5.7. We see that over 61% of responses from students in each group are positive while 11% of responses in each group are negative. The data shows that students in all groups consistently find DISSAV and the hands-on activity engaging.

Table 5.6: Effect of prior experience with program visualization tools on student learning

Experience Level	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
No Experience	19.4%	63.9%	11.1%	2.8%	2.8%	6
Little Experience	28.2%	46.1%	20.5%	2.6%	2.6%	13
Some experience	33.3%	50%	16.7%	0%	0%	7

Table 5.7: Effect of prior program visualization tool experience on student engagement

Experience Level	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
No Experience	16.7%	44.4%	30.5%	2.8%	5.6%	6
Little Experience	30.7%	32%	25.6%	7.8%	3.9%	13
Some experience	23.8%	42.8%	23.9%	9.5%	0%	7

We analyze whether responses to student learning questions are consistent across groups with varying levels of prior C programming knowledge. We summarize the results in Table 5.8. We see that over 76% of responses from students in all groups are positive while only 6% or less of responses are negative. We see that over 83% of responses from students with no experience are positive. We emphasize that perceived learning improves even in students with no prior experience with C programming,

which is in line with our explicit intention of making DISSAV and the hands-on activity accessible to students in this group.

We analyze whether responses to student engagement questions are consistent across groups with varying levels of prior C programming knowledge. We summarize the results in Table 5.9. We see that over 62% of responses from all groups are positive while over 83% of responses from students with no experience are positive. The data shows that students in all groups consistently find DISSAV and the hands-on activity engaging.

Table 5.8: Effect of prior C programming knowledge on student learning

Experience Level	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
No Experience	37.5%	45.8%	12.5%	0%	4.2%	4
Little Experience	19.6%	59%	15.1%	3%	3%	11
Some experience	28.3%	48.3%	23.4%	0%	0%	10

Table 5.9: Effect of prior C programming knowledge on student engagement

Experience Level	Strongly Agree	Agree	Neither A/D	Disagree	Strongly Disagree	N
No Experience	29.2%	54.2%	16.6%	0%	0%	4
Little Experience	15.1%	47%	21.3%	12.1%	4.5%	11
Some experience	36.7%	26.7%	30%	5%	1.6%	10

Overall, our data shows that the majority of students across all four demographic groups that we analyze find DISSAV and the hands-on activity engaging. While the data shows that DISSAV and the hands-on activity consistently improve students' perceived learning (even in students with no prior experience on stack smashing,

program visualization, or C programming), we need more research and data to come to a more definitive conclusion on whether students' learning actually improved.

5.4 Comparative Grade Analysis with Previous Semesters

We compare grade averages from a subset of questions from the Fall 2021 secure software module quiz and class exam with exact or similar questions from the Fall 2019 & Spring 2019 semesters where the stack smashing module was taught with traditional activities (i.e., activities without a guided inquiry learning style and / or program visualization tools). We hypothesize that our program visualization tool and guided-learning activities improve student learning, or the grades of the students, compared to previous semesters. The null hypothesis is that there is no change in student learning or more specifically, there is no statistically significant difference between the grade averages we analyze. The subset of questions we select for the quiz and exam grade comparisons tests students' knowledge on the material we cover in the guided-learning activities, DISSAV and the accompanying hands-on activity. The UNC Charlotte 's Institutional Review Board (IRB) approves the use of data collected from previous semesters.

We select 11 exact or similar questions that we asked in the Fall 2021, Spring 2019, and Fall 2019 secure software module quiz for comparison. We average the grades from the 11 questions for an overall view of the grades. We summarize the results in Table 5.10. While we do not see an improvement in the Fall 2021 semester quiz grades when compared to Spring 2019, the percentages are very similar. We also see that the averages for both sections of the Fall 2021 semester are greater than the average from the Fall 2019 semester. We conduct an unpaired t-test with a significance level of α=5% on the secure software module quiz grades from the Fall 2019 semester and the Fall 2021 Section 1 & 2. We receive a p-value of 0.0452 when comparing the Fall 2021 Section 1 grades to the Fall 2019 grades, based on which we reject the null hypothesis. The result is expected based on the analysis presented in Section 5.3,

since we found that over 60% of students' perceived learning increased. We compare the Fall 2021 Section 2 grades to the Fall 2019 grades to receive a p-value of 0.1040, indicating that the improvement in Fall 2021 Section 2 grades, even though present, is not statistically significant.

Table 5.10: Grade Analysis for Secure Software Quiz

	Spring 2019	Fall 2019	Fall 2021(1)	Fall 2021(2)
Average	68.52%	60.77%	67.14%	66.45%
N	68	71	58	43

We select seven exact or similar questions that we asked in the Fall 2021, Spring 2019, and Fall 2019 secure software exams. We then average the grades for the questions for an overall view of the grades and summarize the results in Table 5.11.

Table 5.11: Grade Analysis for Class Exam

	Spring 2019	Fall 2019	Fall 2021(1)	Fall 2021(2)
Average	71.43%	60.71%	56.62%	59.10%
N	12	68	58	44

We see a decrease in the exam question averages when comparing the Fall 2019 semester to both sections of the Fall 2021 semester. We use an unpaired t-test with a significance level of α=5%. We compare Fall 2019 semester to the Fall 2021 semester section 1 to receive a p-value of 0.4945, showing no statistical significance between the two groups. We compare Fall 2019 semester to the Fall 2021 semester Section 2 to receive a p-value of 0.7229, showing no statistical significance between the two groups. We exclude the comparison to the Spring 2019 semester as the exam was optional; students who were satisfied with their grade did not have to take the exam. We infer that the 12 students who took the exam were well prepared in an attempt to boost their final grade, leading to higher overall averages.

We also analyze the grade averages from six activities included in the secure software module from previous semesters to the six activities included in the Fall 2021 semester. The six activities from previous semesters include three activities that cover a software security warm-up along with the activities "Understand Call Stacks", "The role of `argv` in stack smashing", and "Shellcode, NOP sleds, and Canaries". The six activities in Fall 2021 include five different guided-learning activities (note that we separate Introduction to C into two activities), along with DISSAV's hands-on activity. We summarize the results in Table 5.12. While we see that all activity grades have a high grade average, we find it hard to come to a conclusion on improvements as the grade percentages for the activities vary across each semester.

We see an increase in activity averages when we compare the Spring 2019 activity grade averages to the Fall 2021 section 2 activity grade averages. We hypothesize that the increases in the grade averages are statistically significant, meaning that DISSAV and the guided-activities improve student learning. Our null hypothesis is that there is no statistical significance to the grade increase in activities meaning that there is not strong enough evidence to state whether DISSAV and the guided-learning activities improve student learning. We conduct an unpaired t-test with a significance level of α=5% on the activity grades from the Spring 2019 semester and the Fall 2021 Section 2. We receive a p-value of 0.1007 when comparing the Fall 2021 Section 1 grades to the Fall 2019 grades, leading us to accept the null hypothesis.

Table 5.12: Grade Analysis for Secure Software Activities

	Spring 2019	Fall 2019	Fall 2021(1)	Fall 2021(2)
Average	78%	81%	71%	82%

In regard to the grade analysis, we note four additional aspects. First, we did not conduct a survey about prior experience levels of students in either Spring or Fall 2019. So, there may be variability in the overall background level of students in past

semesters when compared to Fall 2021. In general, there is obviously also variability in the student body in every semester. Second, Fall 2021 was the first semester back on campus for students at UNC Charlotte due to the COVID-19 pandemic. We recognize that while some students are happy to be back on campus, others are apprehensive about coming back on campus. We observe that many students needed time to adjust back to in-person learning, which requires more engagement and participation in the classroom. Third, during the Fall 2021 semester, the instructor significantly reduced the level of direct instruction for the secure software module when compared to previous semesters. This leads us to believe that there may need to be a better balance between direct instruction and solely student-driven guided-learning activities, especially for undergraduate students.

Lastly, we find that some of the guided-learning activities were too long to complete within a class period. The activities *Introduction to C* and *Process Memory Layout* contain 32 and 38 questions, respectively. Even with the separation of the activity *Introduction to C* into two parts, many students were unable to complete the activity within the time period. While the instructor allowed extra time for students to complete the activity, the lack of continuity of group work on the guided-learning activities may have reduced overall engagement and performance.

CHAPTER 6: RELATED WORK

6.1 Guided-Learning Activities

Process Oriented Guided Inquiry Learning (POGIL) is a student-centered, group-learning instructional strategy and philosophy developed through research on how students learn best. [32]. The activities encourage students to work together in self-managed teams. The activities stimulate critical thinking, problem solving and collaboration. POGIL facilitates activities using clear learning objectives, assessment questions and tips [32].

POGIL Activity Clearinghouse (PAC) [43] contains student-centered instructional activities at various stages of development centered around the (POGIL) pedagogy. The purpose is to facilitate the collaboration, peer review, and classroom testing stages associated with creating high-quality materials meeting the standards approved by The POGIL Project.

6.2 Effectiveness of Program Visualization Tools

Rajala et. al [23] evaluates the effectiveness of the ViLLE tool. The paper makes use of a number of pre and post test questions to analyze the tool. The authors find that the tool enhances students' learning regardless of previous programming experience [23]. The study also finds that the tool benefits novice learners more than learners with some previous experience.

Karnalim [38] evaluates the effectiveness of the visualization tool PythonTutor. The authors use a student survey and a quiz to analyze student use of their tool. The authors find that program visualization is a promising tool to assist student learning programming, particularly in Introductory Programming course [38]. They

also conclude that their tool has positively affected the learning process.

Cisar [31] evaluates the effectiveness of the visualization tool Jeliot 3. The research lasted for two school years and 400 students were included [31]. Their analyze finds that Jeliot 3 does have an influence on the process of learning Java.

6.3 Visualizations for buffer overflow attack

Many buffer overflow visualization tools have been developed and deployed to assist in the education of secure programming.

Sasano [25] proposes a visualization tool for detecting when a program overwrites a return address by a buffer overflow attack. The tool provides a gdb visual of the call stack during the execution of a given C program, to assist novice developers in detecting whether a function contains a buffer overflow vulnerability. The user requires background knowledge of memory and gdb to use and understand the outputs of certain commands. The main focus of Sasano's tool is to check if a function contains a buffer overflow vulnerability while DISSAV aims to simulate an attack scenario. Sasano's tool requires background knowledge of gdb, while DISSAV does not.

Zhang [26] et al. proposes an interactive visualization to teach buffer overflow concepts. This tool displays a segment of memory for the user to learn how a buffer stores memory along with how a program overwrites memory. This tool lacks an interactive call stack representation, which is a key focus of DISSAV.

Walker [27] et al. designs a tool to visualize the process address space for teaching secure C programming. Unlike DISSAV, SecureCVisual does not allow the user to conduct a stack smashing attack by using a payload.

Most closely related to our work is the Simple Machine Simulator (SMS) [28], which gives a dynamic visual representation of the stack during program execution. SMS allows the user to step through a C program while viewing the stack and applies rigid rules for mapping source code to memory. The final exercise allows users to overwrite a return address in an attempt to execute code at a different spot in memory. The

instructor predefines the SMS programs and they cannot be changed by the users during the lab, unlike DISSAV which is highly customizable, allowing users to modify the program and the payload during the lab.

Table 6.1: Comparison of visualizations for buffer overflow attacks

	DISSAV	Zhang	Walker	SMS	Sasano
Stack Visualization	Yes	No	Yes	Yes	Yes
Dynamic Payload	Yes	Yes	No	No	No
Attack Scenario	Yes	Yes	No	Yes	No
Code Visualization	Yes	No	Yes	Yes	Yes

In Table 6.1, we compare and contrast DISSAV with the buffer overflow attack visualization tools discussed above, highlighting the main functionalities provided by each tool. To the best of our knowledge, DISSAV is currently the only tool that provides stack visualization, dynamic payload, attack scenario construction, and code visualization.

CHAPTER 7: CONCLUSIONS

We present DISSAV — a web-based, dynamic, interactive program visualization tool to teach stack smashing attacks. DISSAV allows the user to create a program, construct a payload, and execute the program to attempt a simulated stack smashing attack. DISSAV is designed to be easy to access and use even for novice programmers. Our overall aim is to improve student learning and engagement in advanced cybersecurity topics such as stack smashing attacks, as part of an effort to foster a broader and more diverse student body in cybersecurity.

We discuss the deployment and evaluation of a student survey, a collection of guided-learning activities, DISSAV , and an accompanying hands-on activity. We find that DISSAV and the hands-on activity improve over 75% of students' perceived learning while over 60% of students find the tool and hands-on activity engaging. This thesis finds that the majority of the 26 students who consented to the student survey find DISSAV and the hands-on activity an engaging resource in the classroom. While the data shows that DISSAV and the hands-on activity consistently improve students' perceived learning (even in students with no prior experience on stack smashing, program visualization, or C programming), we need more research and data to come to a more definitive conclusion on whether students' learning actually improved. We compare grade averages from prior semesters to the Fall 2021 semester using a subset of questions from the secure software quiz and an exam. While we find statistically significant student learning improvements when analyzing the averages of a subset of quiz questions, we need further research and data to come to a conclusion on whether the guided-learning activities encourage students to discover the more complex idea themselves since we also find no statistically significant improvements in averages

from the exam grades. While we see that activity grades from all semesters have a high grade average, we find it hard to come to a conclusion on improvements as the grade percentages for the activities vary across each semester.

In future work, to increase engagement and interactivity with DISSAV , we plan to incorporate our hands-on activity into *Criminal Investigations*, an interactive, gamified, scalable, web-based framework for teaching and assessing Internet-of-Things (IoT) security concepts [48]. Specifically, we will present a narrative and questions to the student within the Criminal Investigations framework and then have them perform the actual activity steps within the DISSAV web application. We plan to design a narrative describing how an attacker has successfully completed a stack smashing attack on a UNC Charlotte software system and it is the student's responsibility to recreate the attack so that law enforcement and the UNC Charlotte cybersecurity department can gain more knowledge of how the attack occurred.

We plan to improve aspects of DISSAV that students think need improvement, such as window scaling and provide more hints and explanations. We plan to improve the hands-on activity by providing more process memory layout background within the activity before completing the stack smashing attack. We also plan to release DISSAV in future semesters to gain more data on students' engagement and learning with the tool and activity.

REFERENCES

[1] Check Point Technologies, "Check point research: Cybersecurity attacks increased 50% year over year." https://checkpoint.com/.

[2] S. Furnell, "The cybersecurity workforce and skills," *Computers & Security*, vol. 100, 2021.

[3] W. Crumpler and J. Lewis, "The cybersecurity workforce gap." https://www.csis.org/.

[4] C. Hritcu, "Control hijacking attacks." http://citeseerx.ist.psu.edu.

[5] L.-H. Chen, F.-H. Hsu, C.-H. Huang, C.-W. Ou, C.-J. Lin, and S.-C. Liu, "A robust kernel-based solution to control-hijacking buffer overflow attacks," *Journal of Information Science and Engineering*, vol. 27, pp. 869–890, 05 2011.

[6] L. Vaas, "Pulse secure VPNs get quick fix for critical RCE." https://threatpost.com.

[7] S. Gatlan, "Foxit Reader bug lets attackers run malicious code via PDFs." bleepingcomputer.com.

[8] Cloudflare, "What is buffer overflow?." cloudflare.com.

[9] S. Niculaa and R. Zotaa, "Exploiting stack-based buffer overflow using modern day techniques," *Procedia Computer Science*, vol. 160, pp. 9–14, 2019.

[10] J. Leyden, "Research exposes vulnerabilities in IP camera firmware used by multiple vendors." portswigger.net.

[11] T. Seals, "STEM audio table rife with business-threatening bugs." threatpost.com.

[12] A. Hashim, "Vulnerabilities in Cosori Smart Air Fryer could allow remote code execution attack." latesthackingnews.com.

[13] B. Taylor and S. Azadegan, "Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum," *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, pp. 24–29, 2006.

[14] E. Heinsen and C. McDonald, "Program visualization and explanation for novice C programmers," *Proceedings of the 16th Australasian Computing Education Conference*, vol. 148, pp. 51–57, 2014.

[15] D. Budny, L. Lund, J. Vipperman, and J. Patzer, "Four steps to teaching C programming," *32nd Annual Frontiers in Education*, vol. 2, pp. 18–22, 2002.

[16] D. Radosevic, T. Orehovacki, and A. Lovrencic, "New approaches and tools in teaching programming," *20th Central European Conference on Information and Intelligent Systems*, pp. 23–25, 2009.

[17] A. One, "Smashing the stack for fun and profit." https://inst.eecs.berkeley.edu.

[18] B. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97–123, 1990.

[19] P. Guo, "Online Python Tutor: embeddable web-based program visualization for CS education," *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, pp. 579–584, 2013.

[20] J. Helminen and L. Malmi, "Jype - A program visualization and programming exercise tool for python," *Proceedings of the 5th international symposium on Software visualization*, pp. 153–162, 2010.

[21] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with Jeliot 3," *Proceedings of the Working Conference on Advanced Visual Interfaces*, pp. 373–376, 2004.

[22] J. Cross, D. Hendrix, and D. Umphresss, "jGRASP: An integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond," *Journal of Computing Sciences in Colleges*, vol. 23, no. 2, pp. 170–172, 2007.

[23] T. Rajala, M. Laakso, E. Kaila, and T. Salakoski, "Effectiveness of program visualization: A case study with the ViLLE tool," *Journal of Information Technology Education*, vol. 7, pp. 15–32, 2008.

[24] S. Halim, Z.-C. Koh, V. Loh, and F. Halim, "Learning algorithms with unified and interactive web-based visualization," *Olympiads in Informatics*, vol. 6, pp. 53–68, 2012.

[25] I. Sasano, "A tool for visualizing buffer overflow with detecting return address overwriting," *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies*, vol. 2, no. 5, 2016.

[26] J. Zhang, X. Yuan, J. Johnson, J. Xu, and M. Vanamala, "Developing and assessing a web-based interactive visualization tool to teach buffer overflow concepts," *IEEE Frontiers in Education Conference*, pp. 1–7, 2020.

[27] J. Walker, M. Wang, S. Carr, J. Mayo, and C.-K. Shene, "A system for visualizing the process address space in the context of teaching secure coding in C," *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pp. 1033–1039, 2020.

[28] D. Schweitzer and J. Boleng, "A simple machine simulator for teaching stack frames," *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pp. 361–365, 2010.

[29] J. Urquiza-Fuentes and J. A. Velazquez-Iturbide, "A survey of successful evaluations of program visualization and algorithm animation systems," *Transactions on Computing Education*, vol. 9, no. 2, pp. 1–21, 2009.

[30] J. A. Velazquez-Iturbide, I. Hernan-Losada, and M. Paredes-Velasco, "Evaluating the effect of program visualization on student motivation," *IEEE Transactions on Education*, vol. 60, no. 3, pp. 238–245, 2017.

[31] S. Cisar, R. Pinter, and D. Radosav, "Effectiveness of program visualization in learning Java: a Case Study with Jeliot 3," *International Journal of Computers Communications & Control*, vol. 6, no. 4, pp. 668–680, 2011.

[32] POGIL Project Team, "Process-oriented guided inquiry learning." https://pogil.org/.

[33] DBpedia, "About: Call stack." dbpedia.org.

[34] Rodrigo, "How does a nop sled work?." stackoverflow.com.

[35] USNA, "Lesson 8: Buffer overflow attack." usna.edu.

[36] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer, "Architecture support for defending against buffer overflow attacks," *International Conference on Information Technology: Research and Education*, pp. 243–250, 2003.

[37] E. Kaila, T. Rajala, M.-J. Laakso, and T. Salakoski, "Effects of Course-Long use of a program visualization tool," *Proceedings of the 12th Australasian Conference on Computing Education*, vol. 103, pp. 97–106, 2010.

[38] O. Karnalim and M. Ayub, "The effectiveness of a program visualization tool on introductory programming: A case study with PythonTutor," *Communication and Information Technology*, vol. 11, no. 2, pp. 67–76, 2017.

[39] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski, "ViLLE: A language-independent program visualization tool," *Proceedings of the 7th Baltic Sea Conference on Computing Education Research*, vol. 88, pp. 151–159, 2007.

[40] E. Akeyson, M. Sridhar, and H. Ramaprasad, "DISSAV: A dynamic, interactive stack- smashing attack visualization tool," *Colloquium for Information Systems Security Education*, vol. 9, no. 1, 2022.

[41] T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Robling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally, "Evaluating the educational impact of visualization," *Innovation and Technology in Computer Science Education*, pp. 124–136, 2003.

[42] POGIL Project Team, "Effectiveness of process-oriented guided inquiry learning." https://pogil.org/about-pogil/effectiveness-of-pogil.

[43] POGIL Project Team, "POGIL activity clearinghouse." https://pogil.org/pogil-tools/pac.

[44] CS POGIL Project Team, "Process oriented guided inquiry learning in computer science." https://cspogil.org/Home.

[45] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel TSX," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 380–392, 2016.

[46] G. Kc and A. Keromytis, "e-NeXSh: achieving an effectively non-executable stack and heap via system-call policing," *Proceedings of the 21st Annual Computer Security Applications Conference*, pp. 15–302, 2005.

[47] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with PACed canaries," *Proceedings of the 4th Workshop on System Software for Trusted Execution*, pp. 1–6, 2019.

[48] J. Hall, A. Mohanty, P. Murarisetty, N. Nguyen, J. Bahamon, H. Ramaprasad, and M. Sridhar, "Criminal Investigations: An interactive experience to improve student engagement and achievement in cybersecurity courses.," *Proceedings of the 53rd ACM Technical Technical Symposium on Computer Science Education*, 2022.