BEST PAPER AWARD

DISSAV: A Dynamic, Interactive Stack-Smashing Attack Visualization Tool

Erik Akeyson
College of Computer and Informatics
UNC Charlotte
Charlotte, North Carolina
eakeyson@uncc.edu

Harini Ramaprasad
College of Computer and Informatics
UNC Charlotte
Charlotte, North Carolina
hramapra@uncc.edu

Meera Sridhar
College of Computer and Informatics
UNC Charlotte
Charlotte, North Carolina
msridhar@uncc.edu

Abstract—This paper describes *DISSAV: Dynamic Interactive Stack Smashing Attack Visualization*, a program visualization tool for teaching stack smashing attacks. DISSAV is a web-based application built with ReactJS. DISSAV provides a simulated attack scenario that guides the user through a three-part stack smashing attack. Our tool allows the user to create a program, construct a payload for it, and execute the program to simulate an attack scenario. We aim to improve student learning of advanced cyber security topics, more specifically, stack smashing attacks, by increasing student engagement and interaction. We incorporate previously researched techniques of Program Visualization tools such as dynamic user input and interactive views to achieve these goals.

Keywords—DISSAV, Dynamic, Interactive, Stack-Smashing, Attack, Visualization, Programing Visualization, Cybersecurity, Cybersecurity Education

I. Introduction

The global skill shortage in the cybersecurity field is well known by business owners and experts in the field [8]. 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. An estimated three and a half million cybersecurity positions will be unfilled in 2021 [35] due to unavailability of cybersecurity experts. 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 [5].

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 [14, 3]. These types of attacks are quite popular today [34, 9]. A common technique for conducting a control hijacking attack is exploiting a *buffer-overflow* vulnerability [14, 3], a vulnerability that allows an attacker to write data to a buffer that overflows the buffer's capacity, overwriting adjacent memory locations [4]. 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 [22, 2]. Buffer overflow vulnerabilities have the ability to alter video streams from an IP cameras [19], eavesdrop on conversations through desktop conferencing IoT gadgets [30], and even start one'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; stack smashing attacks are representative of control hijacking attacks because they both aim to take control over a system. Buffer overflow attacks, especially 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 [32]. However, teaching stack smashing is a complex task due to the vast background information 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 [23].

Additionally, teaching programming is a difficult task due to its abstraction and complexity [25] and research has shown the C language to be particularly difficult for novice programmers to understand [7]. 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 [1]. Prior work suggests that program visualization is a beneficial resource in the classroom [10, 13, 20, 15, 24, 11]. 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. In this paper, we present *DISSAV: Dynamic Interactive Stack Smashing Attack Visualization*, a web-based, dynamic, interactive program visualization tool for teaching stack smashing attacks. DISSAV guides the user through a stack smashing attack scenario construction through instructional, incremental steps. DISSAV's call stack visualization provides important details such as the call stack growth direction, the layout of an individual stack frame, and movement of data on the stack.

DISSAV's interactive call stack and stack frame aim to increase student engagement. The early 2000s saw a number of influential papers [21] on the engagement of visualization tools which proposed six categories of engagement. DISSAV falls in the *constructing* category, found to be the second highest level of engagement [33]. To the best of our knowledge, DISSAV is the only dynamic visualization of the stack memory that allows the user to replicate a stack smashing attack by constructing a payload.

The main contributions of our work include the following.

- We design and develop DISSAV, an interactive, web-based, stack visualization tool for teaching stack smashing attacks.
- We implement an attack scenario that allows the user to customize vulnerable functions and payloads through dynamic input.

Roadmap Section II provides background information about stack smashing attacks. Section III describes the design of DISSAV. Section IV discusses related work. Section V presents our conclusion.

II. Background: A Stack Smashing Attack

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 [6]. 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 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 [23] 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 [23]. 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 *shell-code* (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 [23]. 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 [26]. 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 [23]. 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 [18].

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 [17, 37].

III. DISSAV

DISSAV is an interactive program visualization tool that aims to teach stack smashing attacks to undergraduate students. 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.

A. DISSAV Workflow

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 Fig. 1. The user can create a local variable or parameter by specifying the name, selecting a data type from a dropdown box, and declaring a value. DISSAV currently supports char, **int**, and **char[]** data types.

Fig. 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 Fig. 2.

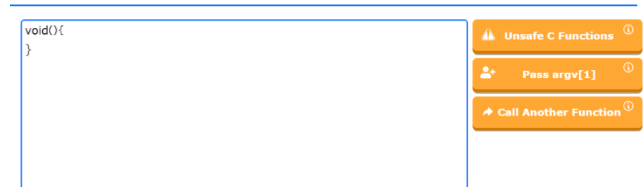

Fig. 2. Function Display

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 Fig. 3. DISSAV dynamically updates the program code as the user adds new (currently up to three) functions.

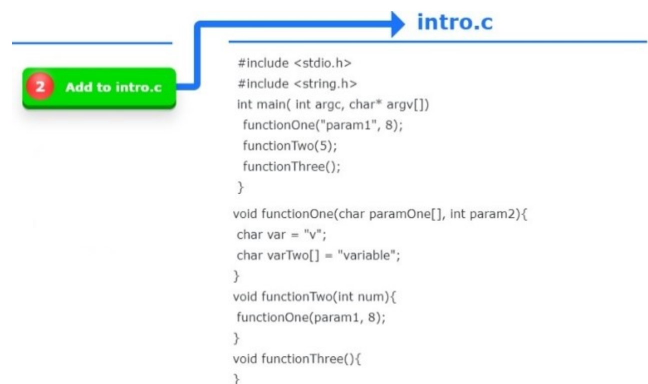

Fig. 3. Program Display

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.

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 Fig. 4. As the user

continues through each part of the payload, DISSAV highlights the corresponding colored section with a border.

4 Construct Payload

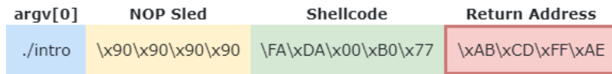

Fig. 4. Dynamic payload diagram

Our payload consists of three parts. Each part contains hints on how to construct the corresponding section, as shown in Fig. 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 II. We implement this design to provide sectioning of the payload, which allows the user to analyze and work on individual pieces to break down each concept.

4.1 Begin with NOP Sled

Hints
* Should only contain \x90
* Consider the space occupied by the local variables
* Return Address and Saved Frame Pointer occupy 4 bytes

\x90\x90\x90\x90

4.2 Add Shellcode

Note
* Be mindful of the length of the machine code

Start a remote shell
\xF3\xDD\xA2\xC9\xAA\xD3
Shut down OS
\xFF\D3\x99\xA0
Get root priviledge
\xCC\xB2\xBB\xA1\x7B\xC8\xF4\xC6
Wipe OS
\FA\xDA\x00\xB0\x77

4.3 End with repeating Return Address

Hints
* Any address that contains a NOP from our payload
* Little endian based CPU
* Repeating occurances of address increases attack success probability

\xAB\xCD\xFF\xAE

Fig. 5. Construct Payload

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 Fig. 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.

Fig. 6. Start Button

The user clicks the **`Next`** button shown in Fig. 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 displays the **`Finish`** button shown in Fig. 8, which pops the main function and ends program execution.

Fig. 7. Next Button

Fig. 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.

a) *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 Fig. 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 III-A3c). 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.

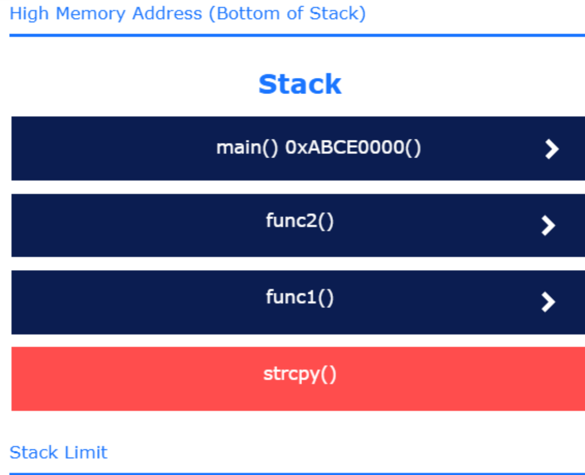

Fig. 9. Call Stack

We do not intend for DISSAV's **Call Stack** to be a detailed program execution call stack similar to ones presented in Jeliot [20], Jype [13], and ViLLE [24], 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**.

b) *Visualize Program Code*: DISSAV highlights the corresponding program line for each movement of a stack frame, as shown in Fig. 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);
}
```

Fig. 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 Fig. 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);
```

Fig. 11. Following argv

c) *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 Fig. 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.

func()		
Parameters	\0	0xABCDFFEE
	r	0xABCDFFED
	a	0xABCDFFEC
	p	0xABCDFFEB
Return Address	\xAB	0xABCDFFEA
	\xCE	0xABCDFFE9
	\x00	0xABCDFFE8
	\x00	0xABCDFFE7
Saved Frame Pointer	\x00	0xABCDFFE6
	\x00	0xABCDFFE5
	\x00	0xABCDFFE4
	\x00	0xABCDFFE3
Local Variables	\0	0xABCDFFE2
	r	0xABCDFFE1
	a	0xABCDFFE0
	v	0xABCDFFDF

Fig. 12. Stack Frame

d) *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 Fig. 13.

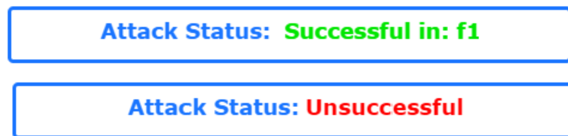

Fig. 13. Attack Status

B. *DISSAV Highlights and Limitations*

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 [21] 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* [33]. We aim to implement *responding* and *presenting* in future work to increase student engagement.

Fig. 14. Landing Page

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 Fig. 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 Fig. 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.

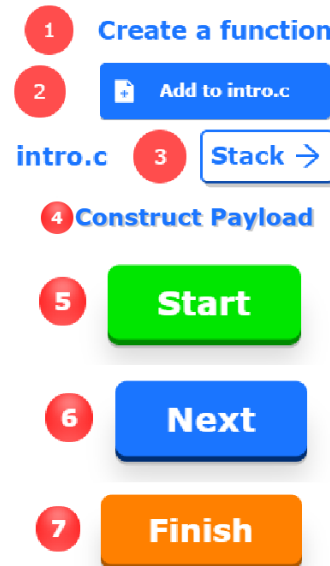

Fig. 15. Instructional Steps

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. Finally, since DISSAV is a web-based application, only users with access to a computer with internet connection can use DISSAV.

IV. Related Works

A. *Program Visualization for Program Execution*

Program visualization is a sub category of software visualization and has been used and researched for decades. Many of these tools aim to improve the education of programming and computing topics to novice programmers. Prior visualization tools [7, 10, 13, 24, 20, 28, 15] provide visual representation of program execution by providing details such as data types, stack frame information, memory, general program flow [16] and source code representation [31]. The effects of these tools are well studied [24, 16] and have been shown to be a beneficial resource in the classroom.

B. *Visualization for Buffer Overflow Attack*

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

Sasano [27] 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 [38] 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 [36] 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) [29], 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.

In Table I, 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.

TABLE I. 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

V. CONCLUSION AND FUTURE WORK

In this paper 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. In Fall 2021, we plan to deploy DISSAV into a software security module of an introductory computer security course.

ACKNOWLEDGEMENT

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

REFERENCES

[1] Brad A.Myers. "Taxonomies of visual programming and program visualization". In: *Journal of Visual Languages and Computing* 1.1 (1990), pp. 97–123.

[2] James C. Foster Vitaly Osipov Nish Bhalla Niels Heinen Dave Aitel. *Buffer Overflow Attacks*. Elsevier Inc, 2005. ISBN: 978-1-932266-67-2.

[3] LI-HAN CHEN et al. "A Robust Kernel-Based Solution to Control-Hijacking Buffer Overflow Attacks". In: Journal *of Information Science and Engineering* 27.3 (2011), pp. 869–890.

[4] Cloudflare. *What is buffer overflow?* URL: https://www.cloudflare.com/learning/security/threats/buffer-overflow/. (accessed: 07.01.2021).

[5] William Crumpler and James Andrew Lewis. *The Cybersecurity Workforce Gap*. URL: https://www.csis.org/analysis/cybersecurity-workforce-gap. (accessed: 04.27.2021).

[6] DBpedia. *About: Call Stack*. URL: https://dbpedia.org/page/Call_stack. (accessed: 06.07.2021).

[7] Matthew Heinsen Egan and Chris McDonald. "Program visualization and explanation for novice C programmers". In: *Proceedings of the Sixteenth Australasian Computing Education Conference* 148 (2014), pp. 51– 57.

[8] Steven Furnell. "The cybersecurity workforce and skills". In: *Computers & Security* 100 (2021).

[9] Sergiu Gatlan. *Foxit Reader bug lets attackers run malicious code via PDFs*. URL: https://threatpost.com/pulse-secure-vpns-critical-rce/166437/. (accessed: 06.04.2021).

[10] Philip J. Guo. "Online python tutor: embeddable web-based program visualization for cs education". In: *Proceeding of the 44th ACM technical symposium on Computer science education* (2013), pp. 579–584.

[11] Steven Halim et al. "Learning Algorithms with Unified and Interactive Web-Based Visualization". In: *Olympiads in Informatics* 6 (2012), pp. 53–68.

[12] Abeerah Hashim. *Vulnerabilities In Cosori Smart Air Fryer Could Allow Remote Code Execution Attacks*. URL: https://latesthackingnews.com/2021/04/27/vulnerabilities-in-cosori-smart-air-fryer-could-allow-remote-code-execution-attacks/. (accessed: 07.15.2021).

[13] Juha Helminen and Lauri Malmi. "Jype – A Program Visualization and Programming Exercise Tool for Python". In: *Proceedings of the 5th international symposium on Software visualization* (2010), pp. 153–162.

[14] Catalin Hritcu. *Control Hijacking Attacks*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.644.7294&rep=rep1&type=pdf. (accessed: 06.06.2021).

[15] James H. Cross II, Dean Hendrix, and David A. Umphresss. “jGRASP: An Integrated Development Environment with Visualizations for Teaching Java in CS1, CS2, and Beyond”. In: *Journal of Computing Sciences in Colleges* 23.2 (2007), pp. 170–172.

[16] Oscar Karnalim and Mewati Ayub. “The Effectiveness of a Program Visualization Tool on Introductory Programming: A Case Study with PythonTutor”. In: *CommIT (Communication and Information Technology) Journal* 11.2 (2017), pp. 67–76.

[17] Mehak Khurana, Ruby Yadav, and Meena Kumari. “Buffer Overflow and SQL Injection: To Remotely Attack and Access Information”. In: (2018), pp. 301–313.

[18] *Lesson 8: Buffer Overflow Attack*. URL: https://www.usna.edu/ECE/ec312/Lessons/host/EC312_Lesson_8_Buffer_Overflow_Attack_Course_Notes.pdf. (accessed: 07.15.2021).

[19] John Leyden. *Research exposes vulnerabilities in IP camera firmware used by multiple vendors*. URL: https://portswigger.net/daily-swig/research-exposes-vulnerabilities-in-ip-camera-firmware-used-by-multiple-vendors. (accessed: 07.13.2021).

[20] Andres Moreno et al. “Visualizing programs with Jeliot 3”. In: *Proceedings of the working conference on Advanced visual interfaces* (2004), pp. 373–376.

[21] Thomas L Naps et al. “Evaluating the educational impact of visualization”. In: *Working group reports from ITiCSE on Innovation and technology in computer science education* (2003), pp. 124–136.

[22] Stefan Niculaa and Razvan Daniel Zotaa. “Exploiting stack-based buffer overflow using modern day techniques”. In: *Procedia Computer Science* 160 (2019), pp. 9–14.

[23] Aleph One. *Smashing The Stack For Fun And Profit*. URL: https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf. (accessed: 05.15.2021).

[24] Teemu Rajala et al. “Effectiveness of Program Visualization: A Case Study with the ViLLE Tool”. In: *Journal of Information Technology Education* 7 (2008), pp. 15–32.

[25] Anthony Robins, Janet Rountree, and Nathan Rountree. “Learning and Teaching Programming: A Review and Discussion”. In: *Computer Science Education* 13.2 (2003), pp. 137–172.

[26] Rodrigo. *How does a NOP sled work?* URL: https://stackoverflow.com/questions/14760587/how-does-a-nop-sled-work. (accessed: 07.15.2021).

[27] Isao Sasano. “A Tool for Visualizing Buffer Overflow with Detecting Return Address Overwriting”. In: *EAI Endorsed Transactions on Self-Adaptive Systems* 2.5 (2016).

[28] Maya Satratzemi, Vassilios Dagdilelis, and Georgios Evagelidis. “A system for program visualization and problem-solving path assessment of novice programmers”. In: *ACM SIGCSE Bulletin* 33.3 (2001), pp. 137– 140.

[29] Dino Schweitzer and Jeff Boleng. “A Simple Machine Simulator for Teaching Stack Frames”. In: *Proceedings of the 41st ACM technical symposium on Computer science education* (2010), pp. 361–365.

[30] Tara Seals. *STEM Audio Table Rife with Business-Threatening Bugs*. URL: https://threatpost.com/stem-audio-table-business-bugs/166798/. (accessed: 07.15.2021).

[31] Lisan Sulistiani and Oscar Karnalim. “An Embedding Technique for Language-Independent Lecturer-Oriented Program Visualization Tool”. In: *EMITTER International Journal of Engineering Technology* 6.1 (2017).

[32] Blair Taylor and Shiva Azadegan. “Threading secure coding principles and risk analysis into the undergraduate computer science and information systems curriculum”. In: *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development* (2006), pp. 24–29.

[33] Jaime Urquiza-Fuentes and J. Angel Velazquez-Iturbide. “A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems”. In: *Transactions on Computing Education* 9.2 (2009), pp. 1–21.

[34] Lisa Vaas. *Pulse Secure VPNs Get Quick Fix for Critical RCE*. URL: https://threatpost.com/pulse-secure-vpns-critical-rce/166437/. (accessed: 06.08.2021).

[35] Cybersecurity Ventures. *Cybersecurity Jobs Report 2018-2021* Edition. URL: https://www.herjavecgroup.com/wp-content/uploads/2018/11/HG-and-CV-Cybersecurity-Jobs-Report-2018.pdf. (accessed: 04.06.2021).

[36] James Walker et al. “A System for Visualizing the Process Address Space in the Context of Teaching Secure Coding in C”. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (2020), pp. 1033–1039.

[37] Jun Xu et al. “Architecture Support for Defending Against Buffer Overflow Attacks”. In: (2002).

[38] Jinghua Zhang et al. “Developing and Assessing a Web-Based Interactive Visualization Tool to Teach Buffer Overflow Concepts”. In: *IEEE Frontiers in Education Conference* (2020), pp. 1–7.