2020-06

STATIC ANALYSIS TOOLS FOR DETECTING STACK-BASED BUFFER OVERFLOWS

Wikman, Eric C.

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

STATIC ANALYSIS TOOLS FOR DETECTING STACK-BASED BUFFER OVERFLOWS

by

Eric C. Wikman

June 2020

Thesis Advisor: Thuy D. Nguyen
Co-Advisor: Cynthia E. Irvine

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2020	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE STATIC ANALYSIS TOOLS FOR DETECTING STACK-BASED BUFFER OVERFLOWS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Eric C. Wikman	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Buffer overflows are common software vulnerabilities; it is possible for a program to write outside of the intended boundary of a buffer. In most cases, this causes the program to crash. In more dangerous situations, a buffer overflow can provide the access an attacker needs to gain remote code execution. To create programs that are reliable and free of buffer overflows, we need a method for analyzing code to detect potential buffer overflow vulnerabilities. One method to detect errors is to perform static analysis on the program. This involves looking at a program's disassembled code to find the errors in the program. Fortunately, Ghidra, a reverse engineering tool, can perform the disassembly of the executable. With the Ghidra API, scripts can be developed to perform the task of analyzing programs for buffer overflows. This research investigates the area of stack-based buffer overflows and how to discover them using static analysis. Specifically, the research looks into cases where buffer overflows occur in libc functions, which are referred to as vulnerable sinks. This research involved the development of a Ghidra script to search for vulnerable sinks in a binary file and find all the parameters that are used in the sinks. This allows for buffer overflows to be calculated on a per-sink basis. The research showed that it is possible to find overflow vulnerabilities via static analysis and that calculating whether a buffer can be overflowed is possible.

14. SUBJECT TERMS buffer overflow, Ghidra	**15. NUMBER OF PAGES** 247
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

STATIC ANALYSIS TOOLS FOR DETECTING STACK-BASED BUFFER OVERFLOWS

Eric C. Wikman
Captain, United States Marine Corps
BS, Auburn University, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2020

Approved by: Thuy D. Nguyen
Advisor

Cynthia E. Irvine
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Buffer overflows are common software vulnerabilities; it is possible for a program to write outside of the intended boundary of a buffer. In most cases, this causes the program to crash. In more dangerous situations, a buffer overflow can provide the access an attacker needs to gain remote code execution. To create programs that are reliable and free of buffer overflows, we need a method for analyzing code to detect potential buffer overflow vulnerabilities. One method to detect errors is to perform static analysis on the program. This involves looking at a program's disassembled code to find the errors in the program. Fortunately, Ghidra, a reverse engineering tool, can perform the disassembly of the executable. With the Ghidra API, scripts can be developed to perform the task of analyzing programs for buffer overflows. This research investigates the area of stack-based buffer overflows and how to discover them using static analysis. Specifically, the research looks into cases where buffer overflows occur in libc functions, which are referred to as vulnerable sinks. This research involved the development of a Ghidra script to search for vulnerable sinks in a binary file and find all the parameters that are used in the sinks. This allows for buffer overflows to be calculated on a per-sink basis. The research showed that it is possible to find overflow vulnerabilities via static analysis and that calculating whether a buffer can be overflowed is possible.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

Listings

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ARCHER	ARray CHeckER
BOON	Buffer Overrun detectiON
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DoD	Department of Defense
MASINT	measurement and signature intelligence
NIST	National Institute of Standards and Technology
NPS	Naval Postgraduate School
ODSS	Overflow Detection from Sinks and Sources
NVD	National Vulnerability Database
NSA	National Security Agency
SRE	Software Reverse Engineering
UNO	Uninitialized variables, nil-pointer dereferencing, and out of bound array indexing
USG	United States government
USN	U.S. Navy

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

First, I would like to thank my advisors Professor Irvine and Professor Nguyen. This was a challenging time, both in the world and in our personal lives. Despite everything, they were always there to lend their knowledge and guidance. I truly appreciate the time and energy they gave to the success of this thesis. Next, I would like to thank my cohort for the camaraderie we built while at NPS. It was always good to know that no matter what the challenge we could always look towards each other for help. Last, I would like to thank my wife and son for being my inspiration and source of joy. Especially my wife, for humoring me when I talked about buffer overflows. Thank you for making this NPS journey an amazing experience.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Buffer overflows are common vulnerabilities in software where an executing program writes to a location outside of the intended boundary of a memory resource (e.g., an array of characters in the stack segment). In most cases, this causes the program to crash. In more dangerous situations, an attacker can exploit a buffer overflow to allow remote code execution. Whether the objective is fixing bugs or creating a secure program, being able to discover buffer overflow vulnerabilities can be exceedingly valuable for developing quality software.

Buffer overflow vulnerabilities can be exploited using a variety of techniques. One method, which is the focus of this thesis, is the occurrence of overflows caused by passing bad parameters to vulnerable sinks. In this context, a sink is a function that receives inputs as execution parameters. Functions that perform little to no bounds checking on their parameters have a potential vulnerability for a buffer overflow. Security researchers often wish to identify the parameters that are supplied to vulnerable sinks. The goal of discovering the parameter values is to determine which combination of parameters could result in an overflow. This process is exceedingly tedious because large programs may contain many vulnerable sinks with many possible sources of parameters. This thesis aims to automate the process of detecting potential stack-based buffer overflow vulnerabilities in binary code using static analysis techniques.

1.1 Motivation

Being able to detect buffer overflows would improve software assurance for the DoD. Static analysis can be used as a baseline for determining the susceptibility of a program to cause buffer overflows. This gives the DoD an idea of the overall quality of the software based on its vulnerability to buffer overflow attacks. The research also benefits the reverse engineering field and automates the discovery of buffer overflows. This could save time and energy for reverse engineers attempting to discover vulnerabilities in a program.

1.2 Research Plan

A stack buffer overflow condition occurs when a variable on the stack is accessed beyond its declared value. This occurs commonly in programming languages such as C or C++, where variables are mapped to memory. Most of the time, these errors are caused by improper input validations within the program. Good input validation checks the type and size of an input before using the input. When these checks do not occur, a program may fail when it tries to access memory locations that are not allocated to the variable. This can occur in functions such as *strcpy()*, where the source string is copied into the destination string. *Strcpy()* does not check the bounds of the input strings and instead uses the null terminator in the source string to know when to stop copying. If the source string is bigger than the destination string, *strcpy()* will write past the end of the destination string, causing a buffer overflow.

This research investigates the viability of a buffer overflow detection method to statically analyze binary files for buffer overflows. The research focuses on the discovery of stack-based buffer overflows that occur in C programs compiled on a Linux x86 64-bit system. Using Ghidra [1], a reverse engineering tool developed by the National Security Agency (NSA) as a foundation, a Python script was developed that uses the Ghidra API [2] to analyze programs for buffer overflow vulnerabilities. To evaluate the method's effectiveness, the National Institute of Standards and Technology (NIST) Juliet test suite [3] is used to determine the error rate of the method.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the background of buffer overflows, related work in buffer overflow detection, and the tools that are used in this work. Chapter 3 provides a high-level overview of the proposed detection method. Chapter 4 discusses the functional requirements and the internal design of the buffer overflow detection program. Chapter 5 explains the detailed implementation of the buffer overflow detection mechanism. Chapter 6 describes our test plan and how the Juliet test suite is used to determine the effectiveness of the proposed method. Chapter 7 analyzes the test results and discusses interesting findings. Chapter 8, the final chapter, summarizes our contributions and provides recommendations for future work.

1.4 Key Terms

The following key terms are used throughout the thesis.

1. Sink: A function that receives inputs to execute its code.
2. Vulnerable sink: A function capable of producing a buffer overflow.
3. Sources: Values or variables that have space or information allocated to them at a particular place inside the program [4].
4. Ghidra: Reverse engineering tool [1].
5. Juliet test suite: A collection of vulnerable programs [3].
6. *Libc*: "Standard library for the C language" [5].
7. Destination parameter: The sink parameter to which values are written.
8. Source parameter: The sink parameter from which values are read.
9. Call location: The address used in the x86 CALL instruction.
10. Reference location: An address that calls or points to another location.
11. Stack location: An address located in the stack memory segment.
12. Source location: A location on the stack that contains the values used for a sink's parameters.
13. Pointer: A location on the stack that contains an address.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter reviews how buffer overflows are caused as well as related research on the discovery of buffer overflows. This chapter also reviews the reverse engineering tool Ghidra [1] and discusses how it can be used to discover buffer overflow vulnerabilities. The last topic covered is the use of the NIST Juliet Test Suite [3] and how it can be used to determine the effectiveness of the buffer overflow detection.

2.1 Buffer Overflow Vulnerabilities

When a buffer is overflowed, it can have severe consequences for the program. Writing past the boundary of a variable by even one byte may corrupt an adjacent variable with bad values. A larger overflow can corrupt a function return pointer. If this is done by accident, the most likely result would be a segmentation fault, because the return call would try to jump to a location to which the program does not have access. Overflow-related bugs in a program also have the potential to allow attackers to overwrite the return pointer with an address of their choosing. This could allow the attacker to gain control of the instruction pointer and execute arbitrary code.

2.1.1 Significance of Buffer Overflows

In 2019 the MITRE Corporation compiled a report of the top 25 most dangerous software errors of the year. The company evaluated the severity of the errors based on the "published Common Vulnerabilities and Exposures (CVE) [7], [8] data and related CWE mappings found within the National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD) [9], as well as the Common Vulnerability Scoring System (CVSS) scores associated with each of the CVEs. A scoring formula was then applied to determine the level of prevalence and danger each weakness presents" [6]. Table 2.1 is a summarized list of the top ten most dangerous software errors from the MITRE report. The number one ranked error is an improper restriction of operations within the bounds of a memory buffer, or, simply put, it is a buffer overflow vulnerability. The scoring was calculated by taking the product of the frequency of the vulnerability and the severity of the vulnerability. The

Table 2.1. CWE Top 10 most dangerous software errors of 2019

Rank	ID	Name	Score
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
[3]	CWE-20	Improper Input Validation	43.61
[4]	CWE-200	Information Exposure	32.12
[5]	CWE-125	Out-of-bounds Read	26.53
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
[7]	CWE-416	Use After Free	17.94
[8]	CWE-190	Integer Overflow or Wraparound	17.35
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	15.54
[10]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10

The MITRE Corporation compiled a list of the most dangerous software errors according to CWEs based on NIST CVE data and CVSS scores. The highest scoring CWE is the result of a buffer overflow. Adapted from [6, Table 1].

buffer overflow vulnerability did not have the highest severity among the top errors, but it was the most frequent vulnerability among the other errors. This indicates that coding a buffer overflow vulnerability is common, and it is hard for developers to catch these types of errors in testing. Having a means to detect this vulnerability during development would help reduce the number of CVEs reported for this type of error.

2.2 Existing Methods that can Detect Overflows

More recently, companies have developed their own proprietary methods for discovering buffer overflow vulnerabilities. In many cases, a company's methods for detecting buffer overflows are proprietary. However, published research exists that describes methods for uncovering buffer overflow vulnerabilities. The following sections discuss a few published methods for detecting buffer overflows.

2.2.1 ARCHER (ARray CHeckER)

ARCHER is a constraint solver tool that checks the bounds of the variables and memory sizes of various objects. ARCHER statically analyzes the source code for memory constraint violations such as array accesses, pointer dereferences, or calls to a function that expects a size parameter [10].

2.2.2 BOON (Buffer Overrun detectiON)

BOON was developed to detect buffer overflows using integer range analysis [11]. This is achieved by parsing through the program looking for string variables. When found, string variables are assigned two integers. The first integer is the string's allocated size, and the second is the number of bytes currently in use. Then, the algorithm checks the usage of each string to determine if the length of the string is greater than the size allocated for the string [11].

2.2.3 UNO (Uninitialized variables, Nil-pointer dereferencing, and Out of bound array indexing)

In other buffer overflow detection techniques, false positives are common. UNO is a tool that aims to reduce the number of false positive reported and to allow users to define their own application-dependent properties for checking for flaws [12]. As its name indicates, the tool checks for errors such as uninitialized variables, nil-pointer dereferencing, and out of bound array indexing. UNO is an extension of a tool called ctree [13], which produces a parse tree of a program. UNO runs the ctree program to get a parse tree, then converts the tree into a control flow graph. Once the graph is created, UNO performs its buffer overflow analysis on the control flow graph. Overall, UNO requires two passes through the code. The first pass is to build the tree and graph, and then check for errors. The second pass performs a global analysis based on the first pass [12].

2.2.4 Value Set Analysis

Value set analysis is the technique of recovering "information about the contents of machine registers and memory locations at every program point in an executable" [14]. This technique was used by Kindermann to perform buffer overflow detection via the static analysis

of executables. This buffer overflow detection method focused on the detection of buffer overflows caused by loops [15].

2.2.5 Machine Learning on vulnerable sinks

Machine learning has been used to predict buffer overflows from vulnerable sinks. Common approaches are to use supervised learning and neural networks to statistically classify overflows. This is done by classifying various elements that contribute to sinks using various machine learning algorithms as implemented in WEKA [16]. The goal of this approach is to determine the probability that a particular sink will cause an overflow. This type of machine learning method has been used on source code as well as binary files by various authors [17]–[20].

2.3 Ghidra

Ghidra is a JAVA-based software reverse engineering (SRE) framework that was developed by the NSA. Ghidra was released to the public in March 2019 and the source code was released the next month on April 4, 2019. Ghidra is the result of about 20 years of development [21]. NSA defines Ghidra as a, "SRE framework developed by NSA's Research Directorate for NSA's cybersecurity mission. It helps analyze malicious code and malware like viruses and can give cybersecurity professionals a better understanding of potential vulnerabilities in their networks and systems" [1].

2.3.1 Ghidra's utility in the discovery of buffer overflows

Ghidra supports the analysis of disassembled code. Ghidra can produce a function database, decompile code, and approximate how local variables are stored on the stack. To build its function database, Ghidra identifies all the functions in the program and resolves the functions that are used in the C standard library (libc). Searching a list of functions allows for the identification of the known *libc* functions that are vulnerable to buffer overflows. The use of a function database makes the detection of buffer overflows easier because it involves searching through a database versus searching through the disassembly.

Approximating the local variables plays a large role in identifying the allocated space on the stack. Without Ghidra, a user would have to look at a disassembled function and

determine how much space is allocated to local variables. This is typically done by looking at the beginning of the function for the SUB RSP, 0x** instruction. However, there are no indications where one local variable starts and another one ends. Instead, the user would have to go through the tedious process of mapping all of a function's stack offsets. Fortunately, Ghidra can perform this analysis to allow users to search all of the found stack references versus searching the function. Once a specific stack reference is known, simple arithmetic can be used to determine the space that is allotted for that reference.

In Ghidra version 9.1-BETA, there are no features that perform the buffer overflow detection task. However, using the Ghidra API, scripts can be implemented to perform various functions such as finding, commenting, and changing functional aspects of the binary file. The research presented here implements a Ghidra script to search a binary file to discover buffer overflows. The script also makes assumptions about the binary file that Ghidra's analysis does not accurately depict. The detailed short falls of Ghidra are discussed in Chapter 5, which also includes a discussion of how they can be overcome.

2.4 NIST Juliet Test Suite

The National Security Agency's Center for Assured Software developed the Juliet test suite and NIST published the test suite [3]. The test suite is described as, "a systematic set of thousands of small test programs in C/C++ and Java, exhibiting over 100 classes of errors, such as buffer overflow, OS injection, hardcoded password, absolute path traversal, NULL pointer dereference, uncaught exception, deadlock, and missing release of resource. These test programs should be helpful in determining capabilities of software assurance tools, particularly static analyzers, in Unix, Microsoft Windows, and other environments" [22]. As of the latest release, there are more than 86,000 programs with known flaws in them. The items in this test suite are categorized according to MITREs CWEs [23]. This thesis uses test cases listed under CWE 121 stack-based buffer overflows [24].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Design Approach

This chapter discusses how static analysis of a binary file can detect buffer overflows in programs. This research investigates cases where buffer overflows occur in *libc* functions. The *libc* functions capable of producing a buffer overflow are referred to as *vulnerable sinks*. In this context, a *sink* is a function that receives inputs to execute its code. The chapter covers how sinks can be found in a program and overviews a process for detecting buffer overflows from the sinks based on their sources. *Sources* are values or variables that have space or information allocated to them at a particular place inside the program [4]. In cases of values, the source may be stored in a register before a function call. In cases of variables, the source may be located on the stack or heap. This work models a technique used by security analysts to manually discover buffer overflows from vulnerable sinks (i.e., tracing the sink's parameters back to their sources). This method, named Overflow Detection from Sinks and Sources (ODSS), allows the analyst to determine if there is a mismatch in parameter sizes that would cause an overflow. The method discussed here is a means of automating the manual process for discovering buffer overflows.

3.1 Primary Causes of Buffer Overflows

In most cases, buffer overflows occur due to improper size validations and missing or incorrect input validation for a given buffer. These can occur in loops when a program indexes past the bounds of an array. They can also occur when functions indiscriminately move values into buffers. Common *libc* functions like *strcpy()*, *strcat()*, *memmove()*, and *fgets()* are known to be susceptible to causing buffer overflows. These functions do not perform validation on their input parameters. These functions are sinks because they receive their information from outside the function. This is not to say that all sink can produce a buffer overflow; it is purely stating that a function that does not check its parameters prior to its execution would be less prepared to handle abnormal input. Looking at an example of *strcpy()*, this function takes two parameters, a destination (parameter one) and a source string (parameter two). *Strcpy()* reads one character at a time from the source string and writes it to the destination string. Once a null byte is read, the null byte is written to the destination

string and the function returns [25]. If the source string is larger than the destination buffer, then the function continues to write past the end of the destination buffer, causing an overflow. *Strcpy()* has no internal mechanism for checking if the destination buffer is greater than or equal to the source string. Thus, it is the responsibility of the programmer to perform those checks prior to calling *strcpy()*. Vulnerable sinks, such as *strcpy()*, are a common entry points for attackers to cause buffer overflows. For this reason, performing analysis on these vulnerable sinks would allow the discovery of potential overflow vectors.

3.2 Detecting Overflows at Vulnerable Sinks

Previous buffer overflow detection techniques, such as Archer, Boon, and Uno, required the size of the variables to be known to perform bounds checking. Although these techniques performed bounds checking in different ways, they followed the general idea that every variable had some space allocated to it and can be filled (initialized) with a string. Bounds checking would compare the allocated size and fill amounts of two variables to see if a buffer overflow is possible. The overflow would occur when the source variable's fill amount is larger than the allocated size of the destination variable. When a source variable's allocated size is larger than the destination's allocated size, it does not guarantee that an overflow would occur, but it does indicate a possible problem. If the fill amount of the source variable were to grow, then the destination variable could be at risk of overflowing.

When looking at source code, finding the parameters' sources is an easier task. The parameter name would resolve to some initialized variable or input parameter for a given function. Initialized variables are clearly stated with the size of the array or the string that is used to initialize the variable. Changes to a variable can be determined by searching for the variable's name, then determining how the variable is used in the instruction. It becomes more complex when sources are passed as parameters, but the actual location of each source can still be determined. To do so would require getting all the references to the called function and checking the variable or value that was used by the calling function. Finding the sinks is similar, because a sink's name can be searched in the program.

When looking at the disassembly of a program, most of the sources and sinks become addresses. When the sources are local variables, they become offsets in the stack. Based on one stack offset alone, it is difficult to determine how much space is allocated to a

particular local variable. To determine the variable's allocated size, the start address of the previous local variable needs to be determined. This allows the difference between the two stack locations to be calculated, which results in the allocated size of the source variable. Parameter passing becomes even more difficult because in 64-bit x86 systems the first six parameters are passed in registers.

3.3 Method for Finding Overflows

As stated earlier, this work aims to automate the manual process that security analysts would use to discover buffer overflows in binary code from sinks. In a typical program, these vulnerable sinks could be called several hundreds of times. This makes that task of verifying vulnerabilities at each sink extremely daunting. Automating this process would significantly reduce the time required to analyze the vulnerable sinks.

This research investigates detecting overflows based on sources and sinks. Unlike Archer, Boon, and Uno, our method looks at disassembly information as opposed to the source code. This also differs from value set analysis, because the focus is on sinks and not loops. Our method consists of four main processing steps: find the sinks, find the sources, determine the source usage, and calculate the overflow. Every CALL instruction to a vulnerable sink has the potential to cause a buffer overflow, so naturally CALL instructions to the sink are the first place to check. At each of the sink calls, the sources are passed to the sink to perform the sink's operation. These source values can come from any location in the program. This means that their actual location on the stack needs to be determined for each parameter passed to the sink. Once all the sources have been collected, they can be analyzed to determine their allocated size and fill amount. Once all the sources have been determined for a given sink, the sources can be compared to determine which combination of sources could cause an overflow.

3.3.1 Finding sinks

Finding vulnerable sinks can be achieved by searching Ghidra's function database, which Ghidra creates by analyzing all functions in the program. Ghidra can also resolve *libc* functions with their actual function names. A search through the database allows for the identification of all the sinks used in the program. Ghidra is also able to create a reference

database, which contains reference information about functions, global variables, initialized data section(s), and the uninitialized data (bss) section. The reference database keeps track of the addresses from which each function is called. The references to the sinks contain the parameters passed to the sink. This means the references to the sinks becomes the starting point to search for the sources.

3.3.2 Finding sources

On 64-bit x86 systems, parameters are passed to functions via registers and the stack. The first six parameters are passed in registers, while any additional parameters are passed on the stack. This requires the detection method to track register usage and stack locations from function calls.

Finding sources starts with the call address to the sink. Since sinks in this research have less than six parameters, the parameters are stored in registers. By tracking the register usage backwards through the code, it is possible to determine values that are loaded into the registers. These values can vary from integer values, local variables, function parameters, or addresses. For integer values, local variables, and addresses, the search ends because these values indicate a location or size, and thus, the source has been found. When source values originate from a parameter or pointer, then the source's true location still needs to be determined. In these cases, all the references to the function or pointer need to be checked. In the case of a function parameter, a path tree is formed where the call to the sink is the root and the functions where the sources are located are the leaves and interior nodes. For the tree to be built properly, repeat functions cannot appear on the same path. This means that the method does not double search functions to prevent loops in the tree. Instead, the method looks at every function once along a particular tree path. Figure 3.1 shows a sink that can have multiple sources used as its parameters. The called sink takes two parameters and for simplicity all the parameters have the same name between functions. Both sink's parameters come from the parameters of Func1. Both Func2 and Func3 call Func1, so they are added to the tree. Func2 is similar to Func1 because both of the parameters in the call come from Func2's parameters. In Func3, par2 is allocated inside the function. For Func4, Func5, and Func6, the parameters relevant to the sink are found inside their respective functions.

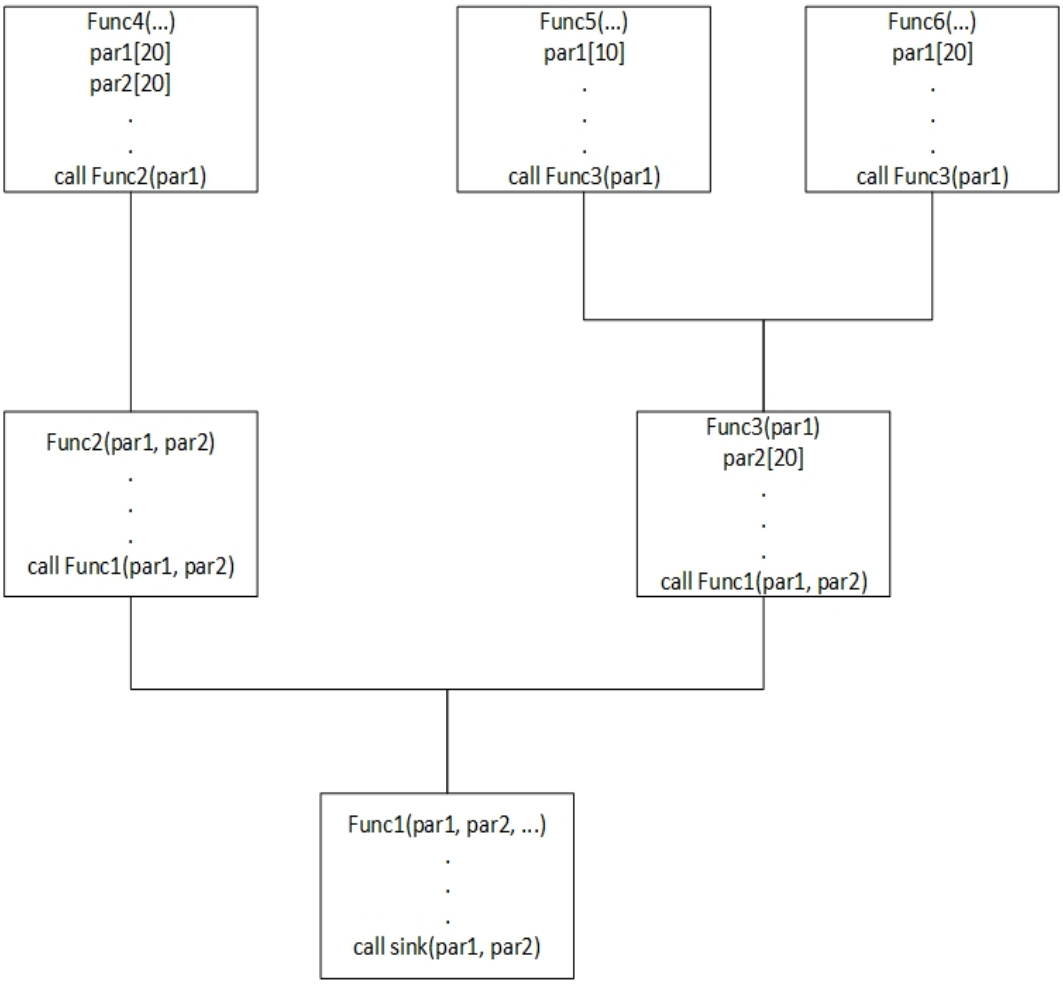


Figure 3.1. Source tree with the sink as the root

3.3.3 Determine the source usage

Once all the sources have been found, the binary file needs to be searched again to understand how the sources are filled. This can be done by following the path from the sources to the sink. Along the way other paths may open depending on the other functions that use the source. Searching involves checking every instance where the source has information written to its location. This accounts for the fill amount of the sources. This is a two-pass process because information about one source could fill in more information about another source.

3.3.4 Calculating the overflow

To calculate overflows all sources for a given sink are compared to determine if a buffer overflow is possible. When the allocated size of a destination parameter is smaller than the allocated size of a source parameter, a caution message is produced. A caution message does not mean that there is an overflow; it only indicates that it is bad practice to copy from a source parameter that is larger than the destination parameter. When the maximum fill amount of a source parameter is larger than the allocated size of a destination parameter, a warning message is produced. The warning message indicates that an overflow is possible based on the static analysis of the program.

CHAPTER 4:
Design

This chapter discusses the functional requirements, the design of the ODSS script, and the high-level approach. The design section describes the major system modules and their interconnections. The program is divided into two modules. The first module finds all sinks used within the binary file and calculates the overflow for each sink. The second module finds the sources for each sink and determines how the sources are used.

4.1 Functional Requirements

The following are the functional requirements for the buffer overflow detection tool.

1. The ODSS script shall run on a Linux x86 64-bit system.
2. The ODSS script shall function as a Ghidra script and use the Ghidra API provided by a selected Ghidra Version.
3. The ODSS script shall be able to detect buffer overflows in C programs that use the following *libc* functions: *strcpy(), strncpy(), strlcpy(), strcat(), strncat(), strlcat(), wcscpy(), wcsncpy(), wcscat(), wcsncat(), memcpy(), memmove(), gets(), and fgets()*.
4. The ODSS script shall be able to detect buffer overflows in C programs that use the GCC compiler in its default settings.
5. The ODSS script shall detect buffer overflows in stack-allocated strings.
6. The plugin shall use the NIST Juliet test suite for functional testing.

4.2 Design Choices

This section discusses the design choices for the functional requirements.

4.2.1 Ghidra Use

Ghidra is a powerful reverse engineering tool that assists in identifying items on the stack and finding object references (e.g., address locations, function locations, and stack locations) within the binary file. The Ghidra API allows for finding functions, references, and iterating through instructions. Ghidra's function finding API allows for the identification of sinks.

Ghidra's reference finding API allows for a concise list of how and where various elements are used in the binary file. Last, Ghidra's iterating API helps determine where instructions begin and end. This makes moving up or down through the binary file significantly easier. This research utilizes Linux-based Ghidra version 9.1-BETA and associated API.

4.2.2 Linux x86 64-bit system

The script is designed to run on a Linux x86 64-bit operating system. An important distinction for running the experiment on an x86 64-bit system versus a x86 32-bit system is the way that parameters are passed. On x86 64-bit systems, the first six parameters are typically passed in registers. On the x86 32-bit system, parameters are passed on the stack. Passing the parameters in registers presents a unique, but solvable, challenge to tracking how values are moved between functions.

4.2.3 Buffer overflow detection used on C programs

Since the buffer overflow detection method is based on vulnerabilities in *libc* functions, it makes sense for the script to only support C programs. Focusing on one programming language allows the research to have consistency across the different tests and allows for proper analysis of the functionality of the detection method.

4.2.4 Compilation requirements

The GCC compiler can optimize the compilation of code in many ways. To maintain consistency, all test programs are compiled using the default GCC configurations with one stipulation regarding the *-fno-builtin* option. This option prevents functions from being inlined with the code. In this context inlined means that the functions are placed directly in the code versus making a CALL instruction to the function. There are a few *libc* functions, like *memmove()*, that are inlined when compiled using the default settings. The exact configurations for GCC can be found in Appendix D. Test cases with multiple files are statically compiled together.

4.2.5 Buffer overflows in stack-allocated strings

There are many ways to overflow a buffer. This research focuses on the overflows that occur from stack-allocated strings. Overflows on the stack are more suited for static analysis

methods. Sizes of the variables that are allocated on the stack are known before run time. When variables are allocated on the heap, their sizes can grow or shrink as the program executes. This makes it difficult for static analysis methods to analyze heap-based variables because the program flow would need to be determined to correctly calculate the space allocated to the variable.

4.2.6 Functional testing using the Juliet test suite

The Juliet test suite has a large volume of vulnerable programs that allows for repeatable testing of the effectiveness of the overflow detection method. The particular set of vulnerabilities that are relevant to this work comes from the section labeled *CWE 121 stack-based buffer overflow*. This set of vulnerabilities exemplifies the type of overflows that this research aims to detect.

4.3 Constraints

This section describes the capabilities that are not in the scope of this thesis and are not implemented

4.3.1 Variadic functions

Variadic functions are functions that take a variable number of parameters. Examples of this from *libc* are *printf()* and *scanf()* [26] [27]. These functions rely on what is known as a format specifier to parse the number of parameters it takes. For this research, these types of functions are excluded from testing. It is not to say that finding a buffer overflow in these functions is impossible. It would require a different approach to parse the format specifier first.

4.3.2 Stack manipulation functions

There are functions that can adjust the stack frame. An example of this is the *alloca()* function. The manual page states, “the alloca() function allocates size bytes of space in the stack frame of the calling function. This temporary space is automatically freed when the function that called alloca() returns to its caller” [28]. Because of the nature of adjusting the stack frame, the *alloca()* function is inlined to the function that calls it. This makes it

difficult to quickly find *alloca()* when searching through the program. This becomes a task of finding the behavior of the *alloca()* function in the code versus determining if a buffer overflow occurred. In short, functions similar to *alloca()* are not included in the buffer overflow tests.

4.3.3 Reversing complex equations

There are situations where a program uses more than basic arithmetic to calculate values. It is important to know those values to figure out what inputs are sent to the sinks. To calculate the values, the equation needs to be discovered. Complex equations would require a sophisticated state machine to create the equation. This research does not cover the development of such a state machine. Instead, this thesis is constrained to solving addition and multiplication equations for input values to the sinks. This research excluded the use of instructions like SUB or DIV when calculating sizes.

4.3.4 Flow invariant

In many cases, programs consist of many conditionals that control the execution of the code. From a static analysis point of view, every possible route through the code would need to be mapped to determine how a program executes. The main goal of this overflow detection method is to test the feasibility of tracking a source back to its origin. For the scope of this research, flow control is not considered. Understanding the control flow of the program can increase the accuracy of the detection method, however being able to map the control flow of a program is a separate large topic [29].

4.3.5 Multiple Register tracking

There are cases where the source's location is calculated based on multiple register values. This is common when indexing into an array. This research implemented single register tracking when finding sources. This is an area that is discussed in future work.

4.3.6 Limited detection of overflow from concatenation functions

Since the control flow of the program is not tracked, knowing when a source had values concatenated to the source is not considered. This thesis investigates overflows caused by single-use overflows, not the continuous modification of a source that could potentially

cause an overflow. A single-use overflow would be to copy 100 bytes into a 50-byte buffer once. A continuous overflow would be to copy 1 byte into a 50-byte buffer 100 times.

4.4 Limitations

Identifying the limitations allows for an understanding of what the overflow detection method cannot do. There are situations that cannot be solved because the problem is considered undecidable. These limitations are not so much to restrict the implementation of this detection method, but to instead identify what was not able to be solved.

4.4.1 Validate sources originating from the argument vector

C programs can take in arguments from the argument vector, *argv*, when the program is executed. These values are stored on the stack before the main function's stack frame. Since this is done at run time, static analysis would be unable to determine the size or the number of parameters passed.

4.4.2 False positives in dead code

If a buffer overflow exists in dead code (i.e., code that is never executed), then it is debatable whether it is a vulnerability if the buffer overflow cannot occur. Since determining if code is dead code is an undecidable problem [30], our overflow detection method is not able to identify if the overflow is in dead code.

4.4.3 Difficulty in detecting overflows in obfuscated code

The whole purpose of obfuscating code is to make it difficult for others to determine the program's intended behavior. As such, code obfuscation can interfere with the normal process of iterating through instructions. Our detection method is designed to analyze binaries created by a GCC compiler. When the assembly code is not generated by the compiler, the detection method is not always able to detect when an overflow can occur. This involves the use of inline assembly code that disrupts the compiler's normal code generation.

4.4.4 Delineating the different data types inside structures

When variables of the C data type *struct* are allocated on the stack, the compiler ensures that there is enough room to hold the data structures. In cases where a struct is declared but uninitialized, the compiler would create a section of the stack that is reserved for that struct. The starting locations of each of the data types inside the struct are unknown, until they are referenced. The Ghidra experiments described in Appendix E show that the location of an uninitialized struct's data types may not be able to be determined if there are not enough clues in the code.

4.5 Functional Behavior

The tool that performs the buffer overflow detection is a Python script used within Ghidra. To run the script, the user must compile a C program using GCC with the compiler settings in Appendix C. To analyze the program, the user must open the binary file in Ghidra and then run the script (review Appendix A on how to run a Ghidra script). The script does not place any restriction on the size of the binary file; it can process any file that Ghidra can load. The size of the largest file tested is one gigabyte. The script runs from start to finish without user interaction.

The script produces two types of outputs. The first type is a console print out displaying the warning or caution messages and the second type is in the form of a CSV file. There are two CSV output files. Even if there are no buffer overflows, the two CSV files are always created. The first CSV file contains specific information for each of the sources, such as its name, location, size, and the sink with which they were used. The second CSV file contains all the sources and sinks that can produce a warning or caution message. Sample outputs from the script can be seen in Appendix F.

In the event that a source cannot be found, that source is dropped from the list of sources. The script does not notify the user when a source is dropped. If the detection script identifies the wrong source, the script may report either a false positive or a false negative.

4.6 Core Modules

This section describes the high-level approach for detecting overflows and an overview of the design of the script, including the specifics of what the modules perform and how

the modules interact. The program consists of three modules: *Main, Sink, and Source*. The Main module is responsible for the program's execution flow. The Sink module is responsible for finding, creating, and calculating the overflows for the sinks. The Source module is responsible for finding, creating, and filling in the information for sources. Figure 4.1 depicts the class diagram for the buffer overflow detection program.

The Sink module consists of two classes: Sink_Handler and the Sinks. The Source module consists of three classes: Source_Handler, Source_Finder, and Sources. The details of each class's variables and methods are discussed in Chapter 5. Methods that have double underscores are private methods and can only be called from within the class. Methods without the double underscore can be called by other classes that have an instance of the class object. Section 4.6.3 describes how these modules interact inside the program.

4.6.1 Sink Module

The Sink module focuses on the discovery, creation, and overflow calculations for the sinks. The sinks are the cornerstone for identifying buffer overflows. So naturally, the first step is to identify where the sinks are in the binary file. The sink objects are created when they are found in the binary file. The overflow calculation occurs after the source information for all the sinks have been determined.

Find Sinks

Without Ghidra, finding sinks would have been a tedious process of identifying all the calls and mapping the call's location in *libc*. This would require knowing the version of *libc* and having maps to each function for each different version of *libc*. Fortunately, Ghidra takes care of the mapping process and can identify which *libc* function is called. This allows the module to search through Ghidra's function database and compare the function names with vulnerable sink names. Once a sink is found, a sink object is created containing the information for the sink.

Calculate Overflow

The sinks have a common parameter format using a source string, a destination string, and sometimes an integer value. The source string is where the information is coming from. The destination string is where the values are written. Using the information that was

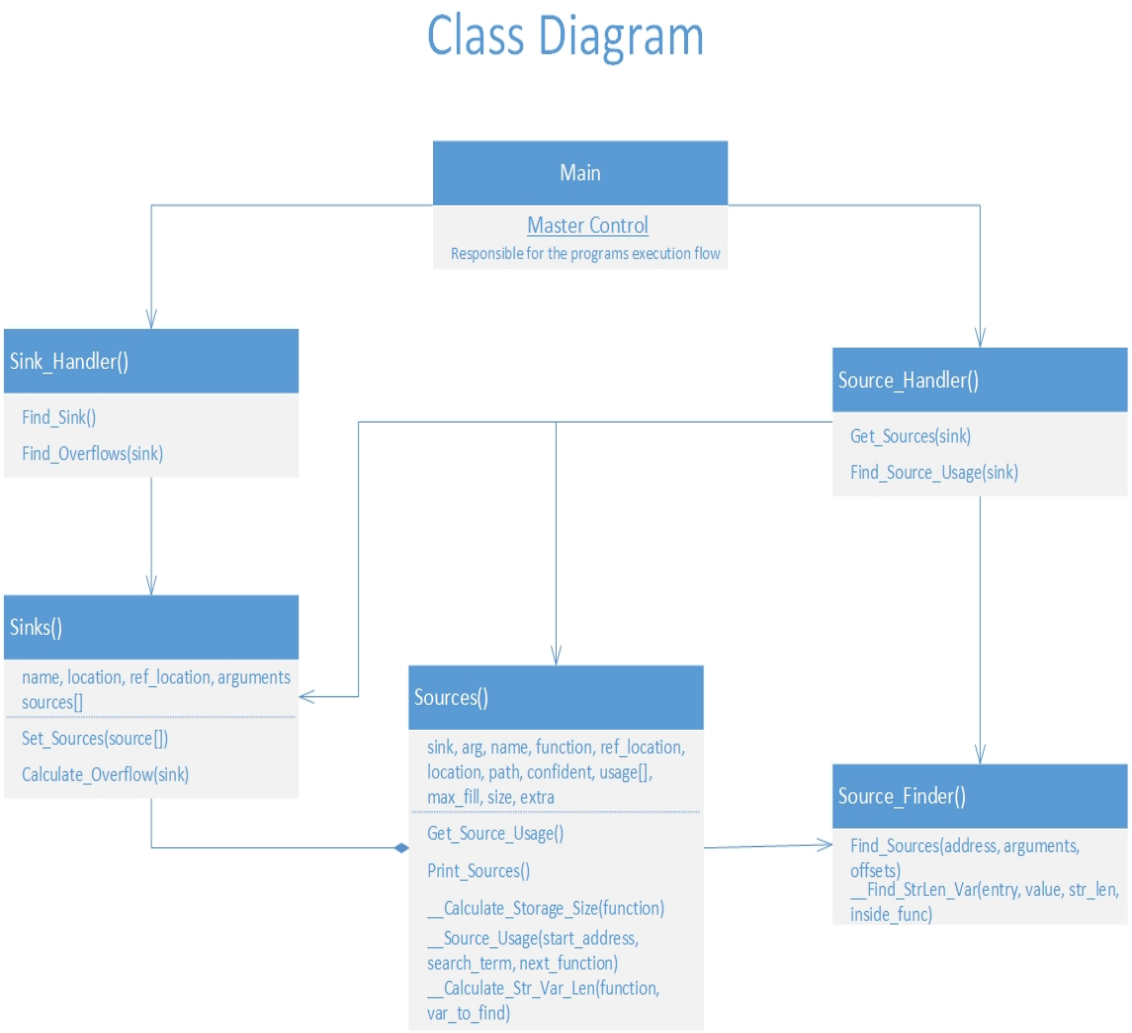


Figure 4.1. Class diagram for the buffer overflow detection program

collected on the sources, a simple range check can determine if the destination string can be overflowed. In the cases of a three-parameter sink, the values are also checked against the integer amount parameter to determine an overflow.

There are three cases that can result from calculating the overflow. The first is no overflow. In this case nothing is reported. The second case is a caution, where the allocated size of the source is larger than the allocated size of the destination, while the maximum fill amount of the source is less than the allocated size of the destination. This condition produces a caution message showing the information of the sources in the sink. The third case is a

warning, which occurs when the source's string length is greater than the allocated size of the destination. This case produces a warning message with the sources that are used at the sink.

4.6.2 Sources Module

This module performs the task of finding the sources and determining the usage of each source. To find the sources, a sink's parameters need to be traced back to its origins. This involves determining how sources are passed to other functions and building the path to each source. Once all the sources have been discovered, the usage of each source can be determined. Searching for the usage of each source allows for the discovery of the longest string length used for the source. This can be performed by checking if information is being written to the source. Knowing the longest string length used in the source allows for the string length to be compared with allocated sizes of other sources to determine if a buffer can be overflowed.

Find Sources

This method takes a sink as input. For each sink, the method determines all the calls that directly reference the sink. These references are known as sink references. The sink references are the first locations checked for finding sources for a particular sink. The Get_Sources method controls the order in which functions are checked for the discovery of sources. The Get_Sources method builds the path with the sink references of a particular sink as the root and subsequent function calls as the leaves and interior nodes.

The next step is to determine how the sources are passed to the sink. The sink references have an address where the sink is called, which is referred to as the entry point. The sink also has a dictionary of parameters to search. From the entry point, the module searches up through the code to determine how the parameters are passed to the sink. Since this tool is for an x86 64-bit system, this logic tracks parameters passed on registers as well as the stack. In the case of static values, addresses, or local variables other than pointers, the module adds these parameters to the sources list. If it is determined that the values were passed to the sink as function parameters, those parameters are used for searching follow-on functions. In the disassembly shown in Figure 4.2, we can see at the bottom of the figure a call to *strcpy()*. For the illustration register RSI, the second parameter to *strcpy()*, is the focus for

tracking. Two lines up from the *strcpy()* call, is a MOV RSI, RDX. This means that RDX holds the value that needs to be tracked. Two more lines up is a MOV RDX, qword ptr [RBP + local_30] instruction. The qword ptr [RBP + local_30] operand is a local variable for this function. Toward the middle of the disassembly, qword ptr [RBP + local_30] is loaded with RDI. This RDI indicates that it was the first parameter to the function. This means that the second parameter to *strcpy()* came from the function's parameter. This would then require all calling functions to be checked to find the source.

Figure 4.2. Ghidra disassembly of a source being passed to a sink

When sources are not found inside a function and it is determined that the source came from

a function's parameter, all the references to said function are found and those referenced locations are searched. Searching the referenced locations is necessary to find all sources that can be used in a sink. This method does not look at the same reference location twice to prevent loops. Each time a new location is searched, it is added to the path that a source took to get to the sink. This process repeats until all the sources have been found or after searching all functions on the path.

Get Source Usage

To accurately calculate the overflows for a sink, the source usage needs to be discovered. This requires a set of methods that performs a detailed analysis of the disassembly code to verify the allocated size and the string length of the sources. Determining a source's usage starts at the location the source was found. From the source's start location, the source is then tracked to determine how it was used, updating its information based on what was found. This involves searching every function that the source is used as a parameter.

For each source, the logic starts off by searching the functions that are on the source's path to the sink. Within each function, the source is checked if it was used in a call to another function. If the source was in a call to another function, then that function would be added to the list of functions to check. To prevent a recursive loop, functions are checked once.

Finding the source usage is performed twice. The first pass is to collect the information that does not have any other dependencies. An example is a static value that can be calculated without further searches. Values that have dependencies could be other sources that have not been calculated yet. The second pass allows the source to calculate values from other sources.

4.6.3 Modules interactions

With a basic understanding of the modules, we can now observe how their components interact. Figure 4.3 shows the sequence diagrams for the buffer overflow detection program. The diagrams show the order the program executes and the interactions between the classes. Function calls from one class to another indicate an interface between classes. Function calls that stay inside the same class are private functions.

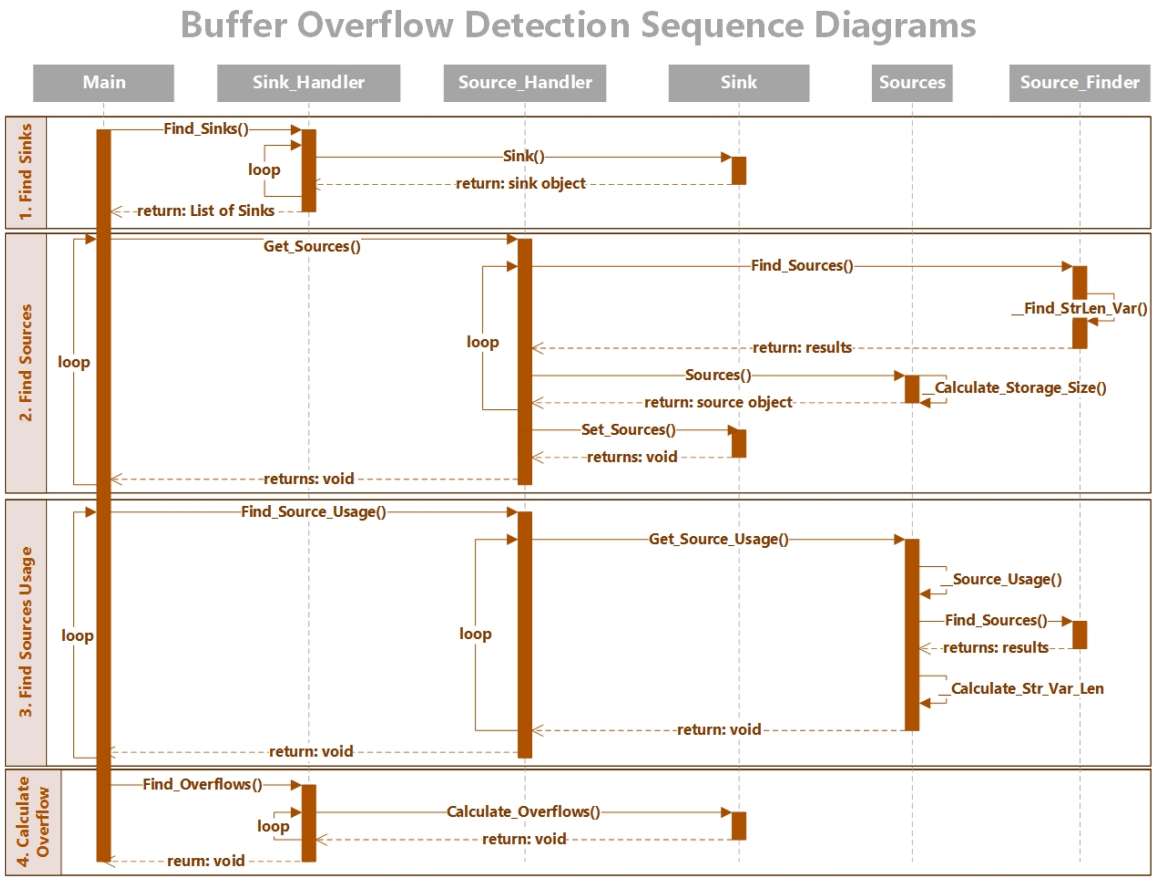


Figure 4.3. Sequence diagrams for buffer overflow detection

1. Find Sinks - sequence diagram 1

The main() function starts the program by calling the Find_Sink method in the Sink_Handler class . Find_Sinks finds the sinks in the program and creates a sink object list, which is returned back to main().

2. Find Sources - sequence diagram 2

The main() function then uses the sinks in the sink list to individually get the sources for each sink. This is done by calling the Get_Sources method in the Source_Handler class. Get_Sources then calls the Find_Sources method in the Source_Finder class to find all the sources associated to the sink. Find_Sources returns the results of its search to Get_Sources, which Get_Sources uses to create a list of source objects. Once all the sources have been found, the sources are then added to the sink object using the Set_Sources method in the

Sinks class. Get_Sources then returns to main(). The main() function repeats the process until all the sinks are processed

3. Find Source Usage - sequence diagram 3

Once the sources for the sinks have been found, main() calls the Find_Source_Usage method in the Source_Handler class for each of the sinks. Find_Source_Usage controls the order that sources are searched and the number of search passes through the binary file. Find_Source_Usage calls the Get_Source_Usage method in the Sources class to conduct the search on an individual source. Get_Source_Usage goes through the program to collect more information about the source. This involves calling other private methods within the Sources class and using the Find_Sources method in the Source_Finder class. In the internal methods of the Sources class, the source information is updated when new information is found. Once complete, Get_Sources_Usage returns to the Find_Source_Usage method. Find_Source_Usage continues this process until two passes through the binary file are complete. Once done Find_Source_Usage returns to main().

4. Calculate Overflow - sequence diagram 4

Once the source usage has been determined, main() calls the Find_Overflow method in the Sink_Handler class. Find_Overflow individually calls the Calculate_Overflow method in the Sinks class for each sink. Calculate_Overflow prints to the console the sinks and sources that can cause an overflow and writes the results to a CSV file. Calculate_Overflow then returns to Find_Overflow. Find_Overflow continues this process for all sinks and then returns to main().

4.7 Summary

This chapter discusses the requirements for the ODSS script as well as constraints for its implementation. The chapter also describes how the modules of the ODSS script work and the sequence in which they are executed. The next chapter further expounds upon the detailed implementation of the ODSS script's classes and methods.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Implementation

This chapter details how the classes and methods perform tasks to find sinks and sources, determine the attributes of a source, and calculate the overflow for a sink. This involves a dissection of the code to view expected inputs and outputs for each class and method. The full source code can be made available upon request. The implementation follows the design concepts discussed in Chapter 4.

The chapter organizes the description of the ODSS script implementation into three sections. The first section describes the global variables and the last two sections discuss the Sink module and the Source module, respectively. The modules-specific sections provide a broad level description of each module's classes and the dependencies the modules have on the Ghidra API. The class subsections list their constructors, parameters, methods, and a description. The method subsections list prototype, parameter, return values, effects, and a description. Figure 5.1 shows the nesting of the modules, classes, and methods.

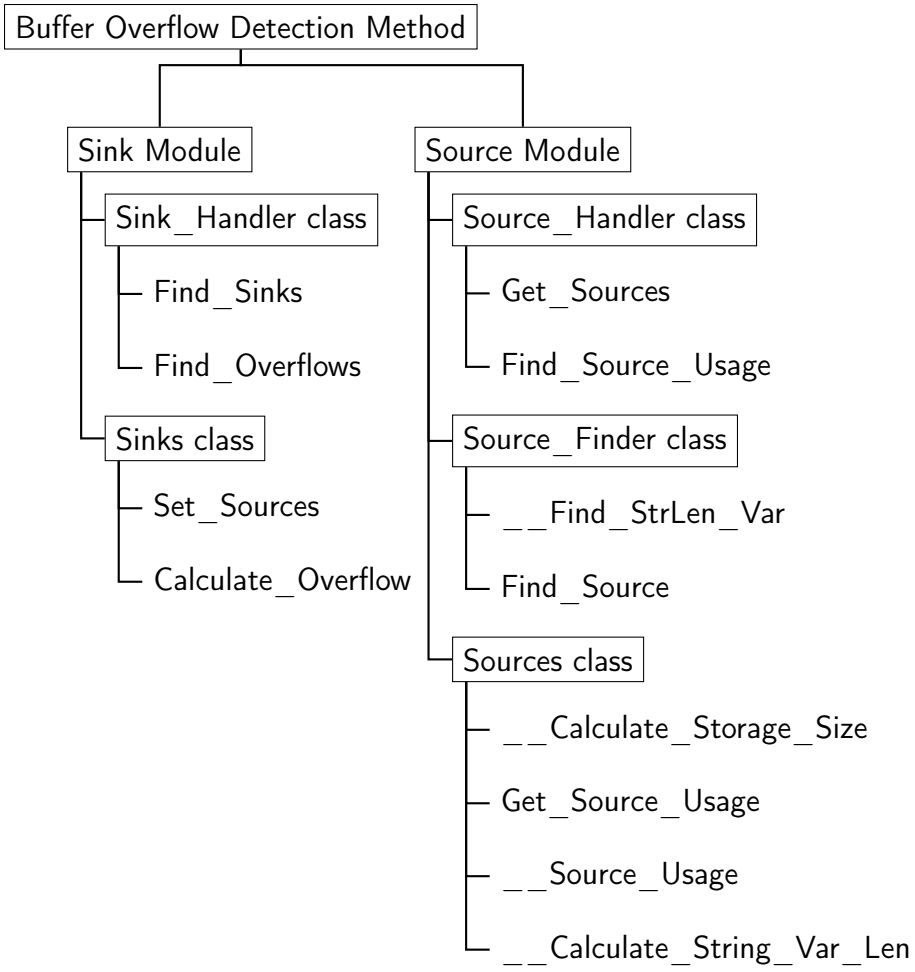


Figure 5.1. Organization of modules, classes, and functions

5.1 Global Variables

The ODSS script uses the following global variables, which are used throughout the script.

- GlobalSources: A list of all the sources found in the binary file.
- FunctionsUsed: A set of all the functions that have been used by the script.
- FunctionLocals: A triple-nested dictionary that contains information on a function's local variables. The first dictionary key is the function's name. The second dictionary key is the local variables offset. The final dictionary consists of three keys: size, type, and block. Size is the number of bytes used in the local variable. Type is the action performed on the variable (e.g., read or write). Block is the number of blocks reserved for this variable.
- SecondPass: A boolean used to indicate if the script is on its second pass through the code.
- OutputFile: A boolean used to indicate if the output file has been created.
- FindSourceLimit: An integer used to limit the maximum level of recursions to be used in the ODSS script.
- Listing: A Ghidra listing object that contains the list of instructions for the binary file.
- AddressFactory: A Ghidra AddressFactory object for referencing addresses in the binary file.
- RefMangager: A Ghidra Reference Manager object for finding references to different objects like functions or variables.
- ProgramName: The name of the binary file.

5.2 Sink Module

The sink module consists of the Sink_Handler and the Sinks classes. The Sink_Handler class is responsible for the discovery, creation, and coordination of the sink objects. The Sinks class is the object that contains the information about the sinks and performs the functions that are specific to a particular sink.

This module depends on the following Ghidra API[1]

- *flatAPI* class
  - getFirstFunction()
  - getReferencesTo(address)
  - getFunctionAfter(function)
- *Function* class
  - getName()
  - getEntryPoint()
- *Reference* class
  - getToAddress()
  - getFromAddress()

5.2.1 Sink_Handler Class

Constructor:

class Sink_Handler()

Parameters:

None

Methods:

Find_Sinks()

Find_Overflow(sink)

Description:

The Sink_Handler class contains two methods. The first method is Find_Sinks. This method performs the discovery and creation of the sink objects. The second method is Find_Overflow. This method sets up the CSV file into which the outputs will be written, and executes the overflow calculation method for the sinks.

[1]http://ghidra.re/ghidra_docs/api/ [2].

Find_Sinks

Prototype:

Sink_Handler.Find_Sinks()

Parameters:

None.

Returns:

sink[]: A list of sink objects.

Effects:

A list of the sink objects is created.

Description:

Find_Sinks is the first core method called by the script. Fink_Sinks takes no arguments and returns a list of sink objects. Find_Sinks uses Ghidra's function database to search for the known vulnerable sinks. Searching is conducted by looking at each function in the database and comparing the function name against the known sink names. In some cases, a function is listed twice in the database. This is handled by adding each new function encountered to a local variable (a set data type). To avoid duplicate entries, the set is checked before each sink is added. Once a sink is found, the references to the sink are gathered using Ghidra's reference API. These references are then used to create the sink objects. The sink object is added to a list that is returned at the end of the function. This is seen in the pseudocode Listing 5.1.

```
function = getFirstFunction()
while function is not None:
    #Filter out non sinks and duplicate functions
    if function in Vuln_Sinks and not in double:
        double.add(function)
        refs = getReferencesTo(entry)
        #Get the references to this sink
        for new_sink in refs:
            sinks_used.append(Sinks(new_sink information))
    function = getFunctionAfter(function)
return sinks_used
```

Listing 5.1: Pseudocode for the primary functionality of Find_Sinks

Find_Overflow

Prototype:

Sink_Handler.Find_Overflow(sink)

Parameters:

sink: Object containing information pertaining to a particular sink.

Returns:

None.

Effects:

Creates the CSV file for storing overflow results.

Description: The Find_Overflow method takes one argument, which is a sink object, and returns nothing. The purpose of this method is to create one output CSV file for the binary file. This method calls Sinks.Calculate_Overflow, using the sinks input object as a parameter. This function uses the *OutputFile* global variable to check if the output CSV file has been created. If it has not been created, the method creates the CSV file and changes the *OutputFile* global variable to true.

5.2.2 Sinks Class

Constructor:

class Sinks(name, location, ref_location, arguments, sources[] = None)

Parameters:

name: The name of the sink as a string.

location: Ghidra address object of the sink's location.

ref_location: Ghidra address object where the sink was referenced.

arguments: Dictionary of the sink's arguments. The key is the argument's position, the value is how the parameter is passed (e.g., charptr1: RDI, charptr2: RSI).

sources: A list of source objects that belong to the sink. This value is an empty list by default.

Methods:
Set_Sources(src[])

Calculate_Overflow(sink)

Description:
The Sinks class contains the information about a particular sink. Sinks in this case are the reference locations that make calls to functions such as *strcpy()* and *memmove()*. The purpose of this class is to organize the information about a sink and perform overflow calculations based on the sink's information.

Set_Sources
Prototype:
Sinks.Set_Sources(src[])

Parameters:
src[]: A list of source objects or an empty list.

Returns:
None.

Effects:
Appends the source list to the sink's data type for sources.

Description:
Set_Sources takes in a list of sources and returns nothing. This method allows the source objects to be assigned to the sink. The method updates the list of sources for a given sink.

Calculate_Overflow
Prototype:
Sink.Calculate_Overflow()

Parameters:
None.

Returns:

None.

Effects:
Writes the sinks and sources that can overflow a buffer to a CSV file and prints the results to the console.

Description:
Calculate_Overflow calculates if a sink can be overflowed by an element from one of its internal list of sources. Since this method is contained within a sink object, this method does not need to take any arguments other than the self object. This method first checks to see what type of sink is used. There are only two types of sinks: a two-argument sink and a three-argument sink. If the sink contains two arguments, the sources are divided into a source parameter list and a destination parameter list. Three-argument sinks contain a source parameter list, a destination parameter list, and an amount parameter list (for the sink's integer parameter). Then the method checks each combination of sources, destinations, and integer amounts parameters (used only in three-argument sinks) to determine if they are on the same path. If the parameters are not on the same path, then those parameters would not end up at the sink together. If they are on the same path, the parameters are compared to determine if there is an overflow. If the maximum fill for the source parameter is larger than the allocated size of the destination parameter, then a warning message is produced and these parameters are written to a CSV file. A warning message is produced for sinks that require three arguments when the maximum fill of the integer amount parameter and the source parameter are larger than the allocated size of the destination parameter. If the allocated size of the source parameter is larger than the allocated size of the destination parameter, then a caution message is produce and the parameters are written to a CSV file. The three-argument case also requires the allocated size of the amount parameter to be larger than the allocated size of the destination parameter. This can be seen in the pseudocode in Listing 5.2.

```
if two argument sink:
    for list of sources:
        group source parameter into list
        group destination parameters into list
    for src in source parameter list
        for dst in destination parameter list
            if src is not on same path as dst:
                continue
            if src.maxfill > dst.size:
                produce a warning message
            elif src.size > dst.size:
                produce a caution message

elif three argument sink:
    for list of sources:
        group source parameter into list
        group destination parameters into list
        group amount parameters into list
    for src in source parameter list
        for dst in destination parameter list
            if src is not on same path as dst:
                continue
            for amt in amount parameter list:
                if amt is not on same path:
                    continue
                if amt.maxfill > dst.size and src.maxfill > dst.size:
                    produce a warning message
                elif amt.size > dst.size and src.size > dst.size:
                    produce a caution message
```

Listing 5.2: Pseudocode for calculating overflows

5.3 Source Module

The source Module consists of three classes: Source_Handler, Sources, and Source_Finder. The Source_Handler class is responsible for coordinating the search for sources, the creation of sources, and the discovery of the source's usage. The Sources class is the object that contains the information about the sources and performs a detailed analysis for each source. The Source_Finder class is responsible for iterating through the code to discover the source location for a given function.

This module depends on the following Ghidra API [2]

- flatAPI class
  - *getFirstFunction()*
  - *getReferencesTo(address)*
  - *getFunctionAfter(function)*
  - *getFunctionBefore(address)*
- Function class
  - *getName()*
  - *getEntryPoint()*
  - *getLocalVariables()*
- Reference class
  - *getToAddress()*
  - *getFromAddress()*
  - *getReferenceType()*
- Listing class
  - *getCodeUnitAt(address)*
  - *getCodeUnitBefore(address)*
  - *getInstructionAt(address)*
  - *getDataAt(address)*
  - *getMaxAddress()*
- Address Factory class
  - *getAddress(string)*
- Address class

[2] http://ghidra.re/ghidra_docs/api/ [2]

  - *next()*
- Instruction class
  - *getMaxAddress()*
  - *getMinAddress()*
  - *getRegister(index)*
- Register class
  - *getParentRegister()*
  - *getName()*
- Data class
  - *hasStringValue()*
  - *getValue()*
  - *getBaseDataType()*
- Variable Class
  - textitgetStackOffset()
  - getLength()

5.3.1 Source_Handler Class

Constructor:

class Source_Handler()

Parameters:

None.

Methods:

Get_Sources(sink)

Find_Source_Usage(sink)

Description:

The Source_Handler class contains two methods: Get_Sources and Find_Source_Usage. Get_Sources performs the discovery and creation of the source objects. Find_Source_Usage coordinates the order in which the source usages are discovered.

Get_Sources

Prototype:

Source_Handler.Get_Sources(sink)

Parameters:

sink: Object containing information pertaining to a particular sink.

Returns:

None.

Effects:

Appends a list of source objects into the sink object.

Description:

Get_Sources is called after all the sinks in the program have been found. Get_Sources takes one argument, a sink object, and returns a list of source objects that belong to a particular sink. The main purpose of Get_Sources is to coordinate the search of the source objects and to build the list of the source objects that belong to a particular sink. This method uses an abstract data structure similar to a stack to build a tree like path from the sink's location to the location that the source originated. The stack initially has one object: the sink that was passed to the function. The sink's information is then passed to Source_Finder.Find_Source to find the sources in that particular function. This returns two outcomes. The first is a list of source information that was found in the function. The source information is then used to create source objects. These sources are then added to a list.

The second outcome is a list of parameters that still need to be found. Get_Sources then finds all the references to the current function and adds them to the stack. When functions are searched, they are popped from the stack. The process of adding functions to the stack and popping the functions that have been searched continues until the stack is empty, meaning there are no more sources that can be found. This is seen in the pseudocode Listing 5.3.

```
stack.append(sink)
while stack:
    #Call function to find sources and values that still need to be
    found
    sources, keep_searching = Find_Sources(stack.pop())
    for src in sources:
        #append source object to source list
        source_list.append(Sources(src information))
    #When elements are in keep_searching, add all the references to the
    #current function to the stack
    if keep_searching:
        refs = getReferencesTo(current_function)
        for new_func_to_search in refs:
            stack.append)(new_funct_to_search)
return source_list
```

Listing 5.3: Pseudocode for the primary functionality of Get_Sources

The following example uses Figure 5.2. Get_Sources uses its sink parameter to get the location of the called sink in Func1. Since not all the sources could be found in Func1, Get_Sources finds all the calls to Func1. In this case there are two calls, one in Func2 and one in Func3. Func2 and Func3 are then pushed on the stack. In this example, we assume that Func2 is now at the top of the stack. This means Func2 is searched next, by popping it from the stack. Since Func2 did not have the sources, Get_Sources then finds all the calls to Func2. This results in finding the call to Func2 in Func4. Func4 is then pushed on the stack. Since Func4 is on top the stack, Func4 is popped from the stack and searched for sources. The search resulted in finding the remaining sources and Get_Sources stops pushing additional functions on the stack. This leaves Func3 on the stack. Func3 is popped from the stack for searching. Get_Sources is notified by Find_Sources that one source was found and one source still needs to be found. Get_Sources then finds the call to Func3 in Func5 and Func6. Func5 and Func6 are then pushed to the stack. The remaining sources are found after these last two functions are searched. This causes the stack to empty because all the sources were found.

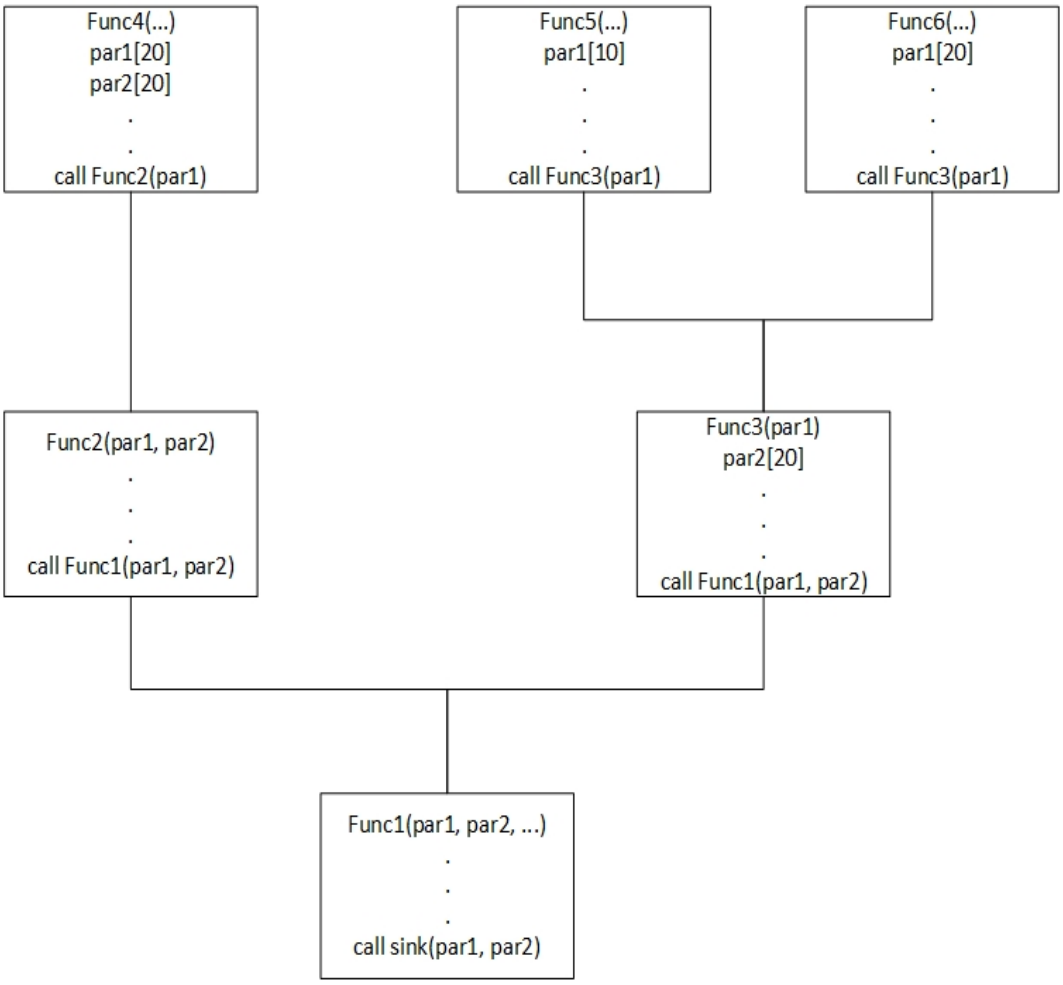


Figure 5.2. Source tree with the sink as the root (repeat from chapter 3)

Find_Source_Usage

Prototype:

Source_Handler.Find_Source_Usage(sink)

Parameters:

sink: Object containing information pertaining to a particular sink.

Returns:

None.

Effects:

Updates the size values for the source objects.

Description:
Find_Source_Usage takes one argument, a sink object, and updates the values for the source objects. This method coordinates the first and second pass for discovering the source's usage.

5.3.2 Source_Finder class

Constructor:
class Source_Finder()

Parameters:
None.

Methods:
Find_Sources(entry_address, entry_args, extra_index)

__Find_Strlen_Var(entry, value, str_len)

Description:
The purpose of the Source_Finder class is to iterate through individual functions in order to determine the source locations. This class contains two methods: Find_Sources and Find_StrLen_Var. Find_Sources searches the given function to discover information about its parameters. Find_Strlen_Var looks at calls to functions such as *strlen()* and *wcslen()* to determine the size of the parameter that was passed to these kinds of functions.

Find_Sources
Prototype:
Source_Finder.Find_Sources(entry_address, entry_args, extra_index)

Parameters:
entry_address: The address location to start searching for the sources as a Ghidra address object.

entry_args{}: A dictionary of the arguments to find. The key is the parameter from the sink. The value is the item (register or stack address) to find (e.g., charptr1: RDI, int: RSI).

extra_index[[argument, operand, value]]: A list of lists containing information used to perform offset calculations to find the source. The outer list consists of zero to many items. The inner list has three items: the sink's argument where the source is used (e.g., the sink's destination parameter is charptr1); a string representation of the value of the operand that contains the base plus offset location (e.g., RSI or [RBP + 0x10]); and the integer value of the required offset.

Returns:
sources[[arg, name, instruction]]: A list of lists containing information about a source location. The inner list has three items: the sink's argument where the source is used (e.g., the sink's destination parameter is charptr1); the name of the location or value of the source as a string (e.g.,: [RBP+-0x40] or 0x45); and an instruction object in which the source was found. The list may be empty if no source locations are found.

args{}: A dictionary of the sources that need to be searched in the calling functions. The dictionary may be empty if all the arguments were found.

index[[argument, operand, value]]: A list of lists containing information used to perform offset calculations to find the sources in the calling functions. The inner list consists of the argument the source was used at the sink (e.g., charptr1). The second item in the list is a string representation of the operand that requires the base plus offset (e.g., RSI). The last item is an integer value of the required offset. The list may be empty if no offsets need to be calculated for the calling functions.

Effects:
Finds source information within a given function and returns the results to the calling function.

Description:
Find_Sources finds source information inside a particular function. Find_Sources takes three arguments: the entry address to start searching, the arguments to search, and any extra offsets that may need to be calculated. Dictionaries are used when there is a unique key to organize items. Lists are used when there is not a unique key. This method returns: a list of sources found in the current function, sources that need to be found in calling functions, and a list of extra offsets that were used in previous functions. Find_Sources consists of several

parts that are used to resolve the sink's parameters to the sources: discovering the functions parameters, handling the extra offsets from the previous function, handling the different instructions, accounting for additional arithmetic to locate stack locations, and determining if a source is a pointer.

Handling Parameters: First, Find_Sources determines the given function's parameters. Ghidra has built-in functions that attempt to find the parameters for a given function, however Ghidra's built-in functions do not always accurately report the true parameters for a function. To overcome this, the Find_Sources method must discover the parameters itself. This is achieved by searching from the beginning of the function. In a majority of cases, the function's parameters are saved as local variables at the beginning of the function. When a register is used to write to a location and the parameter was not used previously, then it is a good indication that the register is a parameter. Searching for the first use of a register allows the script to determine the parameters of the function. The parameters are stored in a dictionary where the local variable is the key and the register is the value. This is seen in the pseudocode listing for finding function parameters (Listing 5.4).

```
while address != end_of_function:
    instruction = getInstruction()
    if store instruction:
        for register in passed_parameter:
            #The register is used, stop looking for parameters
            if register is used:
                reg_used = true
                break
            #this means the register is writing somewhere
            else:
                parameter[location] = register
        if reg_used:
            break
    #gets the next address
    address = address.next()
```

Listing 5.4: Pseudocode for finding function parameters

Handling Extra Offsets: To handle the extra offsets from the previous function, the initial location needs to be found. Once found, the extra offset can be used to determine the actual

stack location for a given source. This is commonly seen with structs, where a pointer to a struct is passed to a function before individual elements in the struct are used. Figure 5.3 shows that only one stack location reference is used for the *strcpy()*. Notice the LEA instruction and how 0x32 was added to the pointer address. This becomes the extra offset that needs to be discovered in the functions that calls this particular function.

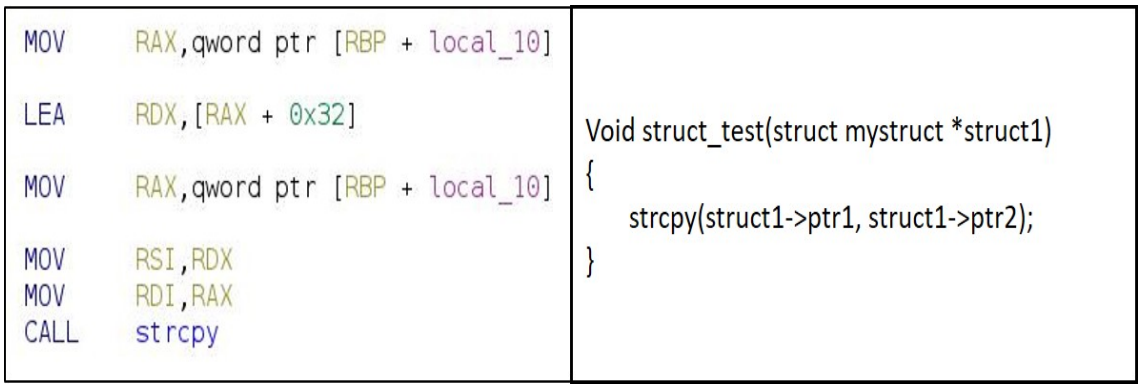


Figure 5.3. The use of offsets from function parameters. Left, disassembly of the *strcpy()*. Right, the C source code.

Handling Instructions: In order to resolve the sources to the sinks, the search begins at the called function. At the called function, the code is searched upwards towards the beginning of the current function. Searching upwards allows for the discovery of the registers and the PUSH instructions that were used in the called function. This process looks at each instruction to determine if the instruction has information about one of the sources. For the purpose of this thesis, the instructions of focus are the data transfer instructions (e.g., MOV, LEA), ADD, PUSH, CALL and JMP. Each will be discussed in detail below.

Data transfer instructions: Data transfer instructions are instructions that perform some type of reading, writing, or both for memory locations. Data transfer instructions, such as MOV, are instructions where a value is read from one location and is written to another location. These instructions require the most scrutiny because of the various values and types that can be used to copy the sources. These instructions check for four cases: sources that are passed to a function by value, sources that came from parameters, sources that require base plus offset calculations, and the source's originating location. This research focused on the following data transfer instructions: MOV, MOVSX, MOVXD, MOVSS,

LEA, and CVTTSS2SI.

When values like structs are passed by value to a function, their values can either be moved or pushed onto the stack. Values that are moved onto the stack require additional calculations to determine its source information. First, it must be determined that the pass by value occurred. This is determined from the previously called function. If the source is not a location in the current function's stack frame (e.g., [RBP + 0x10] is the previous function's stack frame) and is an address (i.e., not a pointer), then the source was passed by value. The pass by value source will have an offset associated with it. When a function saves RBP to the stack, the offset value is subtracted by 0x10 to account for the previous stack frame and saved RBP register. With the adjusted offset, the pass by value source is then compared against the offset that was used to move the source onto the stack. For example, this is seen in instructions similar to MOV qword ptr[RAX], <immediate/register/memory>. When the offsets are equal the source has been found. This was tested against pass by value examples that use RAX to move values on the stack. Using RAX is based on the experiments that were performed in the behavioral tests, which are discussed in Chapter 6. Using RAX is not all encompassing and the register used to move values on the stack may vary depending on the GCC compiler settings.

Table 5.1 and Table 5.2 show an example of values being moved onto the stack for parameter passing. First, the stack is adjusted to make room for the struct. Then piece by piece the values of the struct are moved onto the stack. The beginning of the struct is located at a lower address compared to the end of the struct. Inside the called function, the values of the members in the struct are referenced based on the RBP offset. To access the values, the called function subtracts 0x10 from the RBP offset to account for the saved return and pushed RBP. The resulting offset is compared against the calling function's move location. If a match is found, the particular member is the pass by value parameter. Using the LEA RSI, <immediate/register/memory> instruction in the called function as an example, the value [RBP + 0x18] yields a resulting offset of 0x8. This value is then compared against the MOV instructions from the calling function. The offset from the instruction MOV qword ptr [RAX + 0x8], RDX matches the offset from the called function. This means that the new value to track is the value in the second operand position. In this case the value is the RDX register.

Table 5.1. Instructions for a pass by value parameter

Instructions
SUB RSP, 0x20
MOV RAX, RSP
MOV RDX, Start_Of_Struct
MOV qword ptr [RAX], RDX
MOV RDX, Middle_Of_Struct1
MOV qword ptr [RAX + 0x8], RDX
MOV RDX, Middle_Of_Struct2
MOV qword ptr [RAX + 0x10], RDX
MOV RDX, End_Of_Struct
MOV qword ptr [RAX + 0x18], RDX
CALL Pass_By_Value_Function
Label Pass_By_Value_Function:
PUSH RBP
MOV RBP, RSP
LEA RDI, [RBP + 0x10] #Start_Of_Struct
LEA RSI, [RBP + 0x18] #Middle_Of_Struct1
CALL Some_Function

Table 5.2. Stack view of a pass by value parameter

Stack
End_Of_Struct (High address)
Middle_Of_Struct2
Middle_Of_Struct1
Start_Of_Struct (Low address)
Saved Return
Pushed RBP
(RBP pointer at LEA instruction)

There are cases where the calling function and the called function offsets do not align. This can occur when moving values across two consecutive struct fields. This is most often seen when a character array field is next to another character array field. The end of one character array is moved with the beginning of the next character array. For example, if the program needs to move two four-byte character arrays to the stack, then the compiler may implement one eight-byte MOV instruction to place both arrays on the stack. Two arrays were moved to the stack, however the instruction that placed the arrays on the stack moved both of them at once. Moving both arrays at once makes it difficult to distinguish the end of one array from the start of the other. This can cause the called function's offset to be different from the calling function's offset.

To solve this problem, we had to identify when the calling function's and called function's offsets became unaligned. When searching backwards through the instructions, we find the end of the struct before the beginning of the struct. When the calling offset becomes less than the called offset, this indicates that we passed the beginning of the character array. Thus, when the calling function's offset becomes less than the called function's offset, the previous instruction's offset in the calling function contains the start of the character array. With the start of the array we have now identified the beginning of the source.

Listing 5.5 shows the ODSS script instruction flow for the data transfer instructions. The pseudocode shows how the assembly instructions are inspected to determine how they may affect sources.

The pseudocode starting at line 12 shows the logic to check if a register is being stored with a source. If operand 1 is a register, then the register needs to be checked if it contains a tracked value. If operand 1 is a tracked value, then operand 1 is checked to see if it has a register and an offset. Similar to the extra offsets for previous functions (see handling extra offsets subsection at the beginning of this method), base plus offset arithmetic can be used to calculate locations within the function. The method then checks to see if the offset arithmetic came from a parameter or a local variable. When the operand is a parameter, the value is then added to the extra index list to be tracked in calling functions. When the operand is a local variable, then the base plus offset arithmetic is calculated to get the actual location of the variable. When operand 1 does not have offset arithmetic, operand 2 becomes the new tracked item.

```
if data_transfer_instructions:
    #Check to see if the storing instruction is used to pass by value
    if operand1 was passed by value:
        if source to find location == MOV location:
            track MOV location
        elif  MOV location < source to find location:
            then source to find location is not stack aligned
            Calculate actual offset
            track calculated offset
    #Check to see if a register is being stored with a source
    elif operand1 is a register:
        if register == a source to find value:
            if source to find requires offset math for its location:
                if source to find came from a parameter:
                    add source to find to extra_index list
                elif source to find came from a local variable:
                    calculate actual offset
                    track calculated offset
            #Check if the source value came from a parameter
            if source to find value came from a parameter:
                add to arguments list
            #Does register require a base plus offset calculation
            if operand2 required base plus offset calculations:
                Calculate new offset
                add to extra_math dictionary
                continue
            #Check to see if value is a source
            if operand2 is an address, constant, or local variable:
                add to found_sources list
    else: #The first operand is not a register
        if operand1 == source to find value:
            if source to find came from a parameter:
                add to arguments list
            else:
                if operand2 is an address, constant, or local variable:
                    add to found_sources list
                else:
                    track the operand2
```

Listing 5.5: Pseudocode for data transfer instructions

At this point a tracked value could have changed due to the previous operations. Starting on line 22 of Listing 5.5, when the tracked value changed, it is compared against the function's parameters. If the tracked value came from a parameter, the tracked information is added to the arguments list and then the program continues to the next iteration of the loop.

On line 26 of Listing 5.5, if a tracked value has a base plus offset calculation, then the register becomes the value to track and the offset is saved for when the location is found. When the location for the register is found the offset is added back to the register location to get the actual source location. For this thesis only one register is used to calculate the offset. Future work could include the use of multiple registers for calculating the offset.

On line 32 of Listing 5.5, there is a check to see if the tracked value can be added to the source list. If the tracked value is a local variable, an address, or a constant, then the tracked value is added to the sources list.

Line 36 of Listing 5.5 covers the case where operand 1 is not a register. In this case, operand 1 is checked to see if the operand is a tracked value. If the operand is a tracked value, then operand 2 is checked to see if operand 2 came from a parameter. If it is a parameter the value for operand 2 is added to the arguments list. If operand 2 is not a parameter, then operand 2 is checked to see if it is a source. If operand 2 is a local variable, address, or constant, then the value is added to the source list.

ADD: The ADD instruction indicates that offset calculations are used to point to specific stack values. This is similar to the extra offsets that are passed between functions; however, in this case the values may be local to the function. When an ADD instruction is seen, the operands are checked to see if the instruction affects one of the sources. If a source is affected, that source is added to a dictionary with the amount to add as the dictionary value. This dictionary entry is used later, when the source location is discovered.

PUSH: When parameters are passed by pushing them on the stack, the order in which they are pushed needs to be checked. In the called function, pushed values have positive offsets with respect to RBP. In the calling function, the offset is used to calculate the order in which the value was pushed. This can be achieved by subtracting 0x8 from the offset value, then dividing by 0x8. Subtracting eight accounts for the saved return address. Since the PUSH instruction on 64-bit system always pushes eight bytes, dividing by eight results

in a number that can be used to determine the order with which the value was pushed. The pushes are ordered by their location in the stack. Lower stack addresses have lower push order and vice versa. Table 5.3 shows an example of how the push order is determined. Pushes that are closer to the function call have lower push order values. Knowing the push order, a counter is used to keep track of the number of pushes that have been seen. When the push orders match up, the value that was pushed becomes the new source value to track.

Table 5.3. Calculating push order

Instruction order	Order
Pushed value	Highest push: [RBP + 0x20] = (0x20-0x8)/0x8 = 3
Pushed value	Middle push: [RBP + 0x18] = (0x18-0x8)/0x8 = 2
Pushed value	Lowest push: [RBP + 0x10] = (0x10-0x8)/0x8 = 1
Call Function	

CALL and JMP: JMP/Jcc instructions are indicators of a change in the control flow. A change in control flow may indicate that a source did not take the conditional path to the sink. This problem is outside of the scope of this thesis and potential solutions are discussed as future work in Chapter 8.

In the cases of three-parameter sinks, an integer is needed to determine how many bytes are to be copied. This integer can come from a string length function. When the CALL instruction transfers the execution flow to a function that calculates a string length, the size of the string is returned as an integer. Knowing the result of the string length function allows a better understanding of the number of bytes that can fill the sink. This relies on the Find_StrLen_Var method from the Find_Source_Handler class to determine the integer returned by the string length function.

Resolving pointers: Once iteration through the function to find sources is complete, the sources are examined to determine if they are the actual source location or a pointer. This is done by searching from the beginning of the function to find how the sources are used. If the source is determined to be a pointer, the pointer information is passed back to Find_Source in order to find its true source. This recursive process continues until the pointer resolves to a location. If the pointer was not able to be resolved, then the pointer is passed on as the source. This means the source is misidentified and may result in calculating the wrong

value for the source. Misidentifying the pointer is discussed further in Chapter 7.

Find_Strlen_Var

Prototype:

Source_Handler.__Find_Strlen_Var(entry, value, str_len)

Parameters:

entry: Ghidra address object of the location to start searching.

value: The stack location that is used to calculate the size of a character array in the string length function. This value is passed as a string data type.

str_len: The name of the string length function (e.g., *strlen()*). This value is passed as a string data type.

Returns:

String: Returns a string that contains an equation for solving the size of a given source. If an equation cannot be determined, then an empty string is returned.

Effects:

Calculates the value from functions that return a string's length.

Description:

Find_Strlen_Var is responsible for determining the equation that is used in functions such as *strlen()* and *wcslen()*. This method returns an equation as a string, which is used later once the size of the stack location is determined. This equation is used for determining the maximum fill value for the sink's integer amount parameter. This method starts at the call to the string length function. The method then iterates down through the code to determine the arithmetic that is used on the RAX register (the string length function returns its result in the RAX register). This is performed until the value is loaded into the RSI or RDI register. Registers RSI and RDI indicate that the size of the string is used as the integer parameter at a sink. For this thesis, this method can only solve addition and multiplication problems. Reversing equations that go beyond the use of addition and multiplication is left for future work. This method is not intended to solve multiple uses of string length functions. For example, this will not solve the case of calculating the length of two strings and then adding

them together.

The following example illustrates the use of Find_Strlen_Var. Listing 5.6 shows a sink that uses the *wcslen()* function. Figure 5.4 shows the disassembly of the C source code. The call to *wcslen()* takes in the stack location [RBP + -0x40]. *Wcslen()* returns the result in the RAX register. *Memcpy()* needs the number of bytes, not the wide character length, so RAX is multiplied by 0x4, which is the size of the wide char, to get the number of bytes in this string. The results of the multiplication is loaded into RDX, which is then used in *memcpy()*.

Find_Strlen_Var starts at the call to *wcslen()*. The method then iterates down through the code to determine the arithmetic that is used for the sink. In this case the final equation is ((([RBP+-0x40]/4)+0x1)*0x4). The [RBP+-0x40] is a place holder until the actual size is determined. This location is divided by four because this program always calculates sizes in bytes, not character types. Dividing by four accounts for the wide character type.

```
1 memcpy(dst, src, (wcslen(src) + 1) * sizeof(wchar_t));
```

Listing 5.6: Memcpy() using wcslen()

```
LEA     RAX,[RBP + -0x40]

MOV     RDI,RAX
CALL    wcslen

ADD     RAX,0x1

LEA     RDX,[RAX*0x4]


LEA     RCX,[RBP + -0x40]

MOV     RAX,qword ptr [RBP + -0x48]

MOV     RSI,RCX
MOV     RDI,RAX
CALL    memcpy
```

Figure 5.4. Disassembly of *wcslen()* function

5.3.3 Sources Class

Constructor:

class Sources(sink, arg, name, function, ref_location, location, path, size = None, max_fill = None, usage = None, extra = None)

Parameters:

sink: The name of the sink where the source is used (e.g., *strcpy()*). This value is a string data type.

arg: The parameter of the sink where the source is used (e.g., charptr1). This value is a string data type.

name: A string used as the value or stack location for the source (e.g., [RBP+-0x40] or 0x44).

function: A Ghidra function object where the source was found.

ref_location: A Ghidra address object where the source was called in the sink.

location: A Ghidra address object where the source was discovered.

path[[function, address]]: A list of lists that contains the path from the sink to the source location. The inner list consists of a Ghidra function object and a Ghidra address object.

size: An integer to represent the allocated size of the source. This value is zero by default.

max_fill: An integer to represent the largest fill size of the source. This value is zero by default.

usage[]: A list of instructions as strings that the source was used. This value is an empty list by default.

extra: An equation as a string for calculating the true size of the source. This is used for integer parameters. This value is an empty string by default.

Methods:

__Calculate_Storage_Size(function)

Get_Source_Usage()

__Source_Usage(start_address, search_term)

__Calculate_String_Var_Len(function, var_to_find)

Description:
The Sources class contains the information about a particular source. Sources are the locations/values that are used as parameters in the sinks. The purpose of this class is to search the binary file to determine the sources use and update the information about the source.

Calculate_Storage_Size

Prototype:
Sources.__Calculate_Storage_Size(function)

Parameters:
function: A Ghidra function object that is used to identify the search location.

Returns:
size: Returns an integer that is used to update the value for the source's allocated size.

Effects:
This function updates the object's name, location, function, path, and max_fill values.

Description:
Calculate_Storage_Size is used to obtain the first estimate of the allocated size of the sources. This method estimates the size using information about each source and its surrounding variables.

The method starts by looking at the source's name to check if it is an address or a value. Addresses and values have a similar format: both start with 0x. If the argument for the source is an integer, then values starting with 0x are converted to decimal and the method returns the decimal value as the size. If the value is contained within brackets, then the source is an address, in which case, the method then goes to that address to determine its size. If the value at the location is a string, then the size of the string is returned. If the

value is not initialized, then the references to this location are found. If the reference is a write instruction, then the value to be written is returned for the sources size.

If the source is a local variable, then the adjacent local variables are used to calculate the size. A source is identified as a local variable if it has a stack offset as its name. To estimate the local variable's size, the source's offset is compared against the adjacent variable offset. The adjacent local variables are gathered using Ghidra's API. The difference between the two offsets becomes the initial estimate for the variable's size. Figure 5.4 shows an example stack layout for the local variables. Ghidra produces local variables for every referenced stack location. The first offset is typically a pointer to the stack canary. The remainder of the offsets are variables used by the function. [RBP - 0x10] is eight bytes in length. This could mean that the variable is a pointer, string, or even a double. All we know is that eight bytes of space is available for use. The next variable is at [RBP - 0x20]. This makes 16 bytes of space between the current variable and previous variable, which becomes the estimate of the size of the source. This is only an estimate because there are cases where unallocated structs take up the entire space between variables, but does not indicate the struct's data types. This 16-byte gap could be four-character arrays of four bytes. With the small amount of evidence thus far, we can only assume that the variable used the entire space.

Table 5.4. Estimate of local variable size

Stack	Difference
RBP – 0x8	
RBP – 0x10	0x10 – 0x8 = 0x8
RBP – 0x20	0x20 – 0x10 = 0x10

Get_Source_Usage

Prototype:

Sources.Get_Source_Usage()

Parameters:

None.

Returns:

None.

Effects:
This function updates the values for the size and max_fill values for the source.

Description:
Get_Source_Usage controls the search order for the discovery of a source's information. The method starts at the location the source was discovered and iterates to every function that uses the source. The actual searching of the function is performed by Sources.__Source_Usage. This method updates the source's information when a string longer than the current longest string is written to the source.

Get_Source_Usage starts by adding the source's originating function to an abstract data type similar to a stack. Then the added function is searched for every instance the source was used in a CALL instruction. When the CALL instruction is not a sink, then that function is added to the stack to be searched later. This method does not search the same function parameter pair twice. If the CALL instruction is a sink, then it is determined if the source was the destination parameter. If it is the destination parameter, then the values that were used to fill the source are used to update the source's information. Because sources depend on other sources, Get_Source_Usage is run twice. The first pass is to get the initial information about the source. After all the sources have had a first pass, the second pass is used to fill in gaps about the other sources.

Being able to find the string length of the sources is important for determining if overflows are possible. Figure 5.5 is used as an example for how the source's string length plays a factor in calculating overflows. The figure shows the addition of Func7 and the filling of par2 in Func3. In the case of Func7, we see the need to track where the source is used on the path to the sink. If Func7 was not checked, then par2 would appear to be an empty string. With the information from Func7, we can see that par2 contains 19 'As.' Thus the maximum number of characters that will be copied at the sink is 20 to include the null byte. This example scenario will not cause an overflow at the sink, and if par2 was filled with 20 'As,' there would be an overflow.

When looking at Func3, we see that par2 is filled with nine 'As' with an implied null byte. If we were to only compare the allocated sizes of the sources, we would see that there would

be a false positive from par1 of Func5 (size 10) and par2 of Fun3 (size 20). When we look at the string length for par2, we notice that this does not cause an overflow at the sink, because par2 is only filled with 10 characters including the null byte.

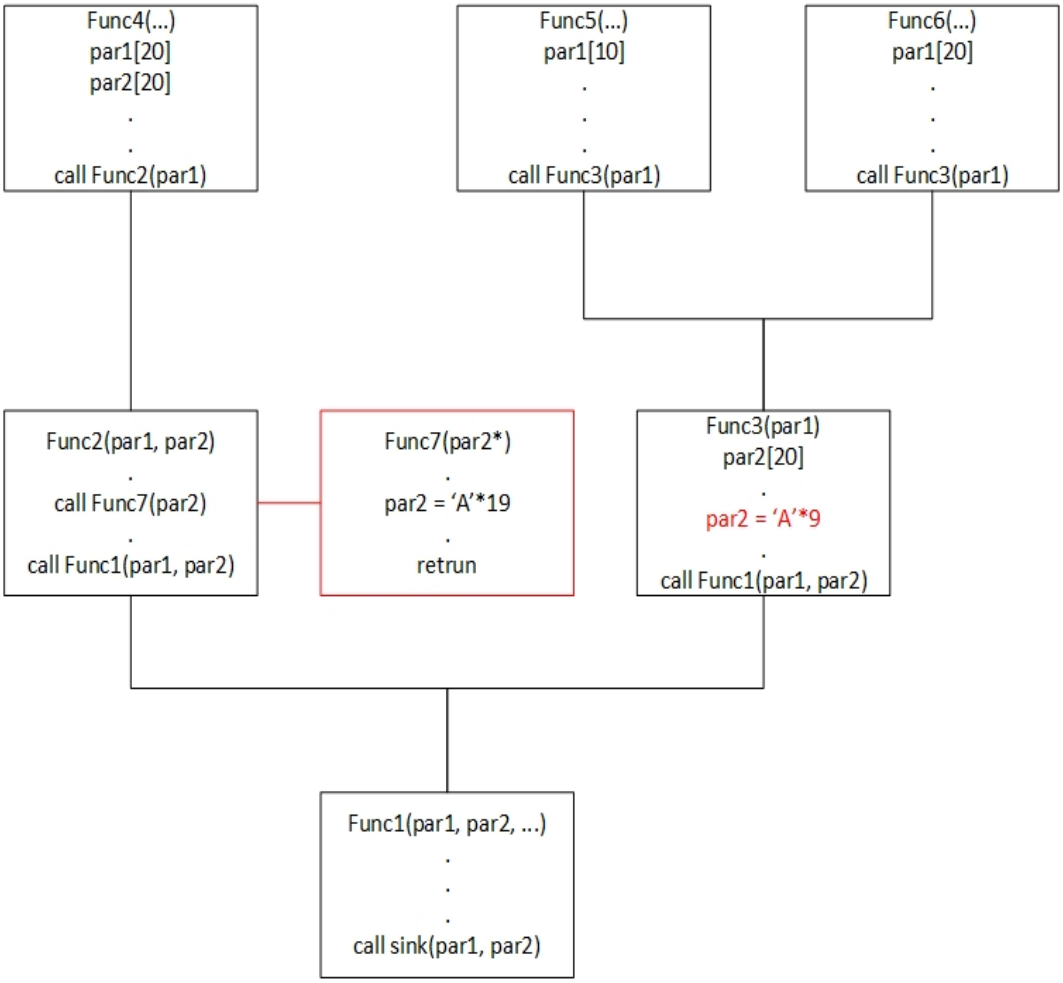


Figure 5.5. Source tree highlighting how the sources are filled

Source_Usage

Prototype:

Sources.__Source_Usage(start_address, search_term)

Parameters:

start_address: A Ghidra address object where the search for the source usage starts.

search_term: A string representation of the value that needs to be found inside the function.

Returns:

Called_func[[function_address, track, instruction_address]]: A list of lists that contains the function calls where the source is used as a parameter. The inner list consists of an address object of the function call, a string representation of the value to track, and an address object containing the location where the CALL instruction was called. This method may return an empty list if the source is not used in a CALL instruction.

Effects:

The method finds information for the sources and updates each source's values based on the findings. It also finds function calls where the sources are used as a parameter.

Description:

Source_Usage searches an individual function for the use of a particular source. This method starts searching from the start_address location looking for every instance that the source is used. This is similar to Find_Sources: it tracks information that is moved around in registers. If the parameter is used in a CALL instruction, then the information about the called function is added to a list. Calls to functions such as *memset()* and *wmemset()* are checked inside Source_Usage to determine the fill amount of the source. By finding the third parameter to *memset()* or *wmemset()*, which tells these functions how many bytes to write to the source, we can calculate the number of bytes written to the source. Knowing the number of bytes written tells us the string length for the source.

Calculate_String_Var_Len

Prototype:

Sources.__Calculate_String_Var_Len(func, var_to_find)

Parameters:
func: The function where the size and max_fill for a source are calculated. This is a Ghidra function object.

var_to_find: A string representation of the value that needs to be found inside the function.

Returns:
None.

Returns:
This method updates the size and max_fill values for the source.

Description:
Calculate_String_Var_Len performs a more in depth estimate of the allocated size and string length of the source. This method returns nothing, but it does update the source's maximum fill and allocated size information. This method determines the size of a variable in a single function. This method is best suited for initialized variables because the method looks at the information that is written to the stack. Uninitialized variables may not have information written to their location until they are used in another function. In this case, information about the uninitialized variables will not be on the stack, which prohibits the ability to determine the string length of the uninitialized source. Although this method does a better job of estimating the source's size information, the algorithm may not accurately estimate the size of the source. This is further discussed in Chapter 7.

The method first finds the fill information of the source. When a local variable is initialized on the stack, its value is moved using a series of operations into the stack memory allocated for that variable. Initializing variables occurs in two ways. First, by moving the bytes individually into the variable's stack location. Second, by using a loop to move values into the variables stack location. When moving bytes individually, the values are moved using the largest segments first, followed by smaller segments. When variables are initialized with a loop, there needs to be a consistent byte value that is moved (e.g., the charater 'a' or the null byte). This is seen in the case of initializing large arrays with the null byte.

To determine the initialized size of the variable, the null byte helps to identify the end of the string, although, in some cases, the null byte may not be present. In those cases, we can

estimate the initialized values that are moved individually by observing where a change in the move sizes occurs. For example, if four bytes were moved followed by eight bytes, it is a good indication that a string ended at the four-byte move. The four-byte move instruction is then checked to see how many bytes were actually moved. When there are three bytes in operand 2, then there may be an implicit null byte at the end to indicate the end of the string. This method does not work for every situation, but it does serve to provide a better estimate of the fill size. An example where this method will not work is when two eight-byte variables are next to each other and are not null terminated. This will cause the variables to look like a string longer than eight bytes.

Figure 5.6 shows the disassembly of a local variable being initialized on the stack. This is an example of a character array of 50 bytes being initialized to contain the 26-byte alphabet. In this case, eight bytes are moved into the first three stack locations. Then, the final two bytes are moved into the following location. The remainder of the bytes are null bytes, which are used to fill out the remainder of the array.

```
MOV     RAX,0x6867666564636261

MOV     RDX,0x706f6e6d6c6b6a69

MOV     qword ptr [RBP + -0x40],RAX
MOV     qword ptr [RBP + -0x38],RDX
MOV     RAX,0x7877767574737271

MOV     EDX,0x7a79
MOV     qword ptr [RBP + -0x30],RAX
MOV     qword ptr [RBP + -0x28],RDX
MOV     qword ptr [RBP + -0x20],0x0

MOV     qword ptr [RBP + -0x18],0x0

MOV     word ptr [RBP + -0x10],0x0
```

Figure 5.6. Disassembly of a local variable being initialized on the stack

The preceding steps calculates the allocated size and the largest string length of the source.

With the largest string length of the source, the method then resolves the equation that was created by the Find_StrLen_Var method for this particular source. The largest string length of the source is then replaced with the variable in the equation. This then allows the equation to be solved and the result of the equation is then used as the largest size of the source. Note that this is only applied to integer parameters.

5.4 Summary

This chapter has built upon the design presented in Chapter 4 by providing detailed information about the implementation of our tool. This involved taking a deeper look into the classes and methods that are implemented inside of the Sink and Source modules. Next we describe the behavioral testing, the purpose of which is to demonstrate how Ghidra was tested to inform the development of the script and the functional test which is used to evaluate the effectiveness of the script.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Test Plan

The test plan consists of two parts. First, the behavioral test verifies how Ghidra can analyze the binary file. The behavioral test is aimed at understanding outputs and syntax Ghidra produces from the disassembly. Second is the functional test for determining the accuracy of the detection method. The functional test utilizes a sample of programs from the Juliet test suite [3].

6.1 Behavioral Test

The behavioral test consists of various function calls and variable initialization to observe how Ghidra analyzes the binary code. With an understanding of Ghidra's capabilities and limitations, refinements can be made to the buffer overflow detection method. The test involves creating local variables of various types, global variables of various types, and structs of strings and an integer. The variables range from being uninitialized, initialized, and initialized to zeros to observe how Ghidra translates these variables. The test then observes the various ways parameters are passed to functions. This involves passing zero parameters, fewer than six parameters, more than six parameters, passing pointers, passing structs, and passing variables by value. The final test observes how Ghidra analyzes casting in a function. The full behavioral tests can be seen in Appendix E.

6.2 Functional Tests

The buffer overflow detection method is tested on ten different sinks with a total of 40 tests, seen in Table 6.2. The functional tests follow the guidance of NIST Special Publication 800-142 Practical Combinatorial Testing by performing pairwise combinations of the test configurations [31]. At the top level, test cases are categorized based on the overflow variant types, which are listed in Table 6.1. Each overflow variant type has a set of flow variant types. The flow variant types are grouped based on the flow type (control flow or data flow) and by the way parameters are declared. A detailed description of flow variant groups is listed in Appendix B. Sinks are divided into two groups: two-parameter sinks and three-parameter sinks. Wide character types and normal character types can be grouped together

because the source's size calculation is performed by counting the bytes at the location of the source. This means the differences between wide or normal characters will not skew the tests. The grouping of overflow variant types, flow variant types, and the number of parameters at the sink forms a hierarchy which can be seen in Figure 6.1.

Table 6.1. Overflow variant types

Title	Description
Struct overrun	Incorrect length value used for a struct
CWE 193	CWE 193 refers to an off by one error
CWE 805	CWE 805 refers to buffer access with incorrect length value
CWE 806	CWE 806 refers to buffer access using size of source buffer
Destination declared	Destination parameter passed to sink as a pointer
Source declared	Source parameter passed to sink as a pointer

The overflow variant types came from the Juliet test suite in the CWE 121 Stack Based Buffer Overflow category [3]

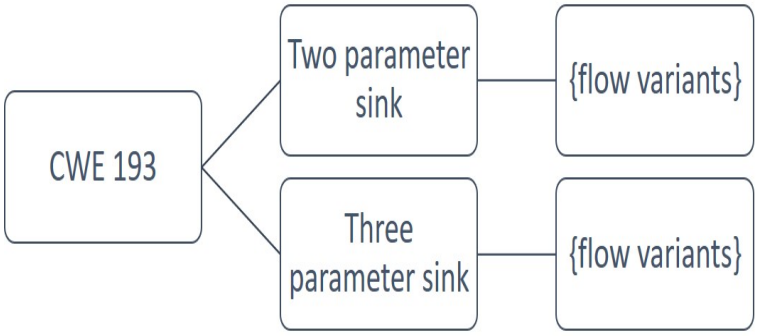


Figure 6.1. Example hierarchy of test cases

The Juliet test suite follows a standard naming convention for its files. The first part is the CWE ID followed by the shortened CWE entry name. This is proceeded by the functional variant name and the flow variant. The functional variant name is considered the flaw type

and the flow variant is a number to represent the control flow used in the program (refer to Appendix B for the flow variant list). The last identifier is the programming language.

Example:

CWE Entry ID: 121

Shortened CWE Entry Name: Stack Based Buffer Overflow

Functional Variant: CWE805_char_declare_memcpy

Flow Variant: 07

Language: C

This creates the file:

CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_07.c

Table 6.2. Summary description of the test cases

Test case	Overflow Variant	Number of Parameters	Flow Variant	Sink
1	Struct overrun	3	1	memcpy()
2	Struct overrun	3	8	memmove() (wide char)
3	CWE193	2	2	strcpy()
4	CWE193	2	31	wcscpy()
5	CWE193	2	34	strcpy()
6	CWE193	2	41	wcscpy()
7	CWE193	2	51	strcpy()
8	CWE193	2	63	wcscpy()
9	CWE193	3	3	memcpy()
10	CWE193	3	32	memmove()
11	CWE193	3	34	strncpy()
12	CWE193	3	44	wcsncpy()
13	CWE193	3	52	memcpy() (wide char)
14	CWE193	3	64	memmove() (wide char)
15	CWE805	3	10	strncat()
16	CWE805	3	31	wcsncat()
17	CWE805	3	34	memmove()
18	CWE805	3	44	memcpy() (wide char)

Table 6.2. Summary description of the test cases

Test case	Overflow Variant	Number of Parameters	Flow Variant	Sink
19	CWE805	3	53	strncpy()
20	CWE805	3	65	wcsncpy()
21	CWE806	3	11	wcsncpy()
22	CWE806	3	22	strncpy()
23	CWE806	3	32	memcpy() (wide char)
24	CWE806	3	34	wcsncat()
25	CWE806	3	42	strncat()
26	CWE806	3	54	memmove()
27	CWE806	3	66	memcpy()
28	Destination declared	2	17	strcat()
29	Destination declared	2	31	wcscat()
30	Destination declared	2	34	wcscpy()
31	Destination declared	2	41	strcpy()
32	Destination declared	2	51	wcscat()
33	Destination declared	2	67	strcat()
34	Source declared	2	18	wcscpy()
35	Source declared	2	21	wcscat()
36	Source declared	2	32	strcpy()
37	Source declared	2	34	strcat()
38	Source declared	2	42	wcscat()
39	Source declared	2	54	strcat()
40	Source declared	2	68	strcpy()

Table 6.2. The test cases came from the Juliet test suite in the CWE 121 Stack Based Buffer Overflow category [3]

6.3 Expected Outputs

The first step in the functional tests is to compile the individual test case. Details on how to compile the test cases can be found in Appendix C. Once the binary for a test case is created,

the binary file is loaded into Ghidra, where Ghidra runs its analysis on the binary. Then, a manual inspection of the disassembly is conducted. Based on the disassembly information, the sinks are evaluated to see if they could be overflowed.

The manual inspection involves locating the sinks and all the sources for the sinks. This requires identifying the space allocated on the stack and the largest string length for each source. This is compared against the source file to determine if there are differences between the source file's allocated space and the disassembly's allocated space. Differences can occur for various reasons with the most common reason involving alignment. If an array is 14 bytes in the source code, the compiler may align the array to the stack. When this happens, the compiler could allocate 16 bytes on the stack to make the array stack-aligned. This essentially makes the array 16 bytes in length. Once all the source values have been determined, they are compared against each another to determine if an overflow is possible.

Listing 6.1 shows the vulnerable source code for test case 15. The manual inspection involves identifying the sinks, sources, and calculating the overflow. In this case, there is one sink: *strncat()*. The function's input parameters are a pointer named data which points to a 50-byte buffer, a char array named source which is 100 bytes in size, and an integer of 100. We can also see that the source parameter is filled with 99 C's from *memset()* and is then null terminated. Based on the source code, the destination parameter is going to be overflowed by 50 bytes.

```
1  void ...CWE805_char_declare_ncat_10_bad(){
2      char * data;
3      char dataBadBuffer[50];
4      char dataGoodBuffer[100];
5      if(globalTrue){
6          data = dataBadBuffer;
7          data[0] = '\0'; /* null terminate */
8      }
9      char source[100];
10     memset(source, 'C', 100-1);
11     source[100-1] = '\0';
12     strncat(data, source, 100);
13 }
```

Listing 6.1: Vulnerable source code

Figure 6.2 shows the vulnerable disassembly for test case 15. This figure shows the stack layout as well as the decompiled code for the test case. In this case, the destination parameter originates from local_b8 and the source parameter originates from local_78. In both the stack layout and the decompiled code, we can observe that local_b8 contains 64 bytes and local_78 contains 99 bytes (the null byte gets its own variable). Based on the disassembly there is an overflow of 36 bytes.

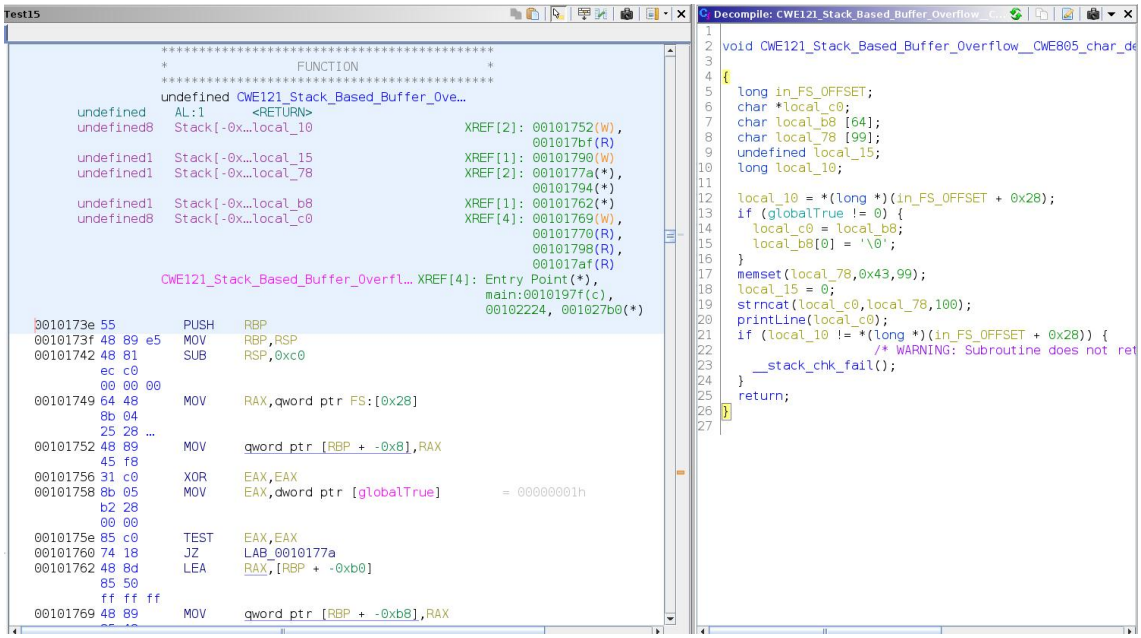


Figure 6.2. Vulnerable disassembly code

Table 6.3 shows the summary of the expected results from the tests based on the manual inspection. To eliminate any confusion between the disparity between the source code and disassembly, Table 6.3 shows the overflow amount as seen in the source code and as seen in the disassembly. Ultimately, determination of whether a buffer overflow exists depends upon the disassembly, not the source code. Thus, the expected results column shows the program's outcome based on the disassembly. The expected results column shows the expected number of messages, the expected type of message, and the expected function name in which the overflow is expected to occur. In the Juliet test cases, functions are marked good or bad depending on their vulnerability at the sink. There are cases where functions labeled good can produce caution messages. In these cases, the function does not change the allocated size of the buffers, but it does reduce the string length in the buffer. The details

of each case can be seen in Appendix G.

Test Case	Overflow Variant	Source code overflow amount	Disassembly overflow amount	Expected results
1	Struct over-run	16 bytes	16 bytes	One warning in bad function
2	Struct over-run	16 bytes	16 bytes	One warning in bad function
3	CWE 193	1 byte	1 byte	One warning in bad function
4	CWE 193	4 bytes	0 bytes	One caution in bad function One cautions in good function
5	CWE 193	1 byte	1 byte	One warning in bad function
6	CWE 193	4 bytes	0 bytes	No overflow
7	CWE 193	1 byte	0 bytes	No overflow
8	CWE 193	4 bytes	0 bytes	No overflow
9	CWE 193	1 byte	1 byte	One warning in bad function
10	CWE 193	1 byte	1 byte	One warning in bad function
11	CWE 193	1 byte	1 byte	One warning in bad function
12	CWE 193	4 bytes	0 bytes	No overflow
13	CWE 193	4 bytes	0 bytes	No overflow
14	CWE 193	4 bytes	0 bytes	No overflow
15	CWE 805	50 bytes	36 bytes	One warning in bad function
16	CWE 805	200 bytes	192 bytes	One warning in bad function
17	CWE 805	50 bytes	36 bytes	One warning in bad function
18	CWE 805	200 bytes	0 bytes	No overflow
19	CWE 805	50 bytes	0 bytes	No overflow
20	CWE 805	200 bytes	0 bytes	No overflow
21	CWE 806	200 bytes	200 bytes	One warning in bad function Two cautions in good function

Test Case	Overflow Variant	Source code overflow amount	Disassembly overflow amount	Expected results
22	CWE 806	50 bytes	50 bytes	One warning in bad function Two cautions in good function
23	CWE 806	200 bytes	200 bytes	One warning in bad function One caution in good function
24	CWE 806	200 bytes	200 bytes	One warning in bad function One caution in good function
25	CWE 806	50 bytes	50 bytes	One warning in bad function One caution in good function
26	CWE 806	50 bytes	50 bytes	One warning in bad function One caution in good function
27	CWE806	50 bytes	50 bytes	One warning in bad function One caution in good function
28	Destination declared	50 bytes	36 bytes	One warning in bad function
29	Destination declared	200 bytes	192 bytes	One warning in bad function
30	Destination declared	200 bytes	192 bytes	One warning in bad function
31	Destination declared	50 bytes	0 bytes	No overflow
32	Destination declared	200 bytes	0 bytes	No overflow
33	Destination declared	50 bytes	0 bytes	No overflow
34	Source declared	200 bytes	192 bytes	One warning in bad function One caution in good function

Test Case	Overflow Variant	Source code overflow amount	Disassembly overflow amount	Expected results
35	Source declared	200 bytes	192 bytes	One warning in bad function Two cautions in good function
36	Source declared	50 bytes	36 bytes	One warning in bad function One caution in good function
37	Source declared	50 bytes	36 bytes	One warning in bad function One caution in good function
38	Source declared	200 bytes	192 bytes	One warning in bad function One caution in good function
39	Source declared	50 bytes	50 bytes	One warning in bad function One caution in good function
40	Source declared	50 bytes	50 bytes	One warning in bad function One caution in good function

Table 6.3. Results of the manual inspection of the source code and disassembly

6.4 Calculating Error Rates

The script produces two types of outputs. The first is a print out to the Ghidra console of all the caution and warning messages for sinks that were detected as vulnerable to buffer overflows. The script also creates two CSV files. The first CSV file contains all the sources that were found in the program. The second CSV file contains sinks that were detected as vulnerable to buffer overflows.

When the destination parameter allocated space is less than the maximum string length of the source parameter, a warning message is produced and the information about the sources is displayed. In the case of the three-parameter sinks, the script stipulates that the

number of bytes to read must be larger than the destination parameter's allocated space. A caution message is produced when the destination parameter's allocated space is less than the allocated space of the source parameter. If a warning message is produced for a given set of sources, no caution message is produced for the same sources.

True positives are counted when a caution or warning message is produced for a sink that is vulnerable to a buffer overflow. A false positive is counted when a caution or warning message is produced when it is not possible for the sink to be overflowed. A false negative is counted when no caution or warning message is produced for a sink that is vulnerable to a buffer overflow. Since actual stack-allocated sizes and the sizes listed in the source code may differ, the differences in byte calculations are not subtracted from the error rates of the predictions.

CHAPTER 7:
Testing Results

The behavioral test gave valuable insight into how Ghidra handles functions and variables. Correctly identifying variable locations is crucial for determining the exact location of the sources. The information gained from this test was used in the implementation of the buffer overflow detection method. The functional testing revealed the feasibility of using this buffer overflow detection method to determine vulnerable sinks. This testing also identified areas where the detection method can improve and identified a limitation in locating sources.

7.1 Behavioral Tests

The main observations from the behavioral tests were the variable labeling and the stack layout produced by Ghidra. Ghidra labels the local variables according to their offset in the function's stack frame, not the RBP offset. For example, when a local variable is labeled local_10, the variable starts at the 16th byte in the stack frame. If RBP was pushed to the stack at the beginning of the function and then set to RSP, the reference to local_10 would be [RBP + -0x8]. For consistency in designing the script, it was important to use the actual offsets and not the local variable name that Ghidra gives to the variables.

The first time a reference is made to a stack frame location, Ghidra would create a local variable for that reference. The local variable would contain the variable's size and a list of references to that variable. The size indicates the amount that was read from or written to the variable. If the size is four, this may indicate a four-byte integer. If the size is one, then this may indicate a one-byte character. If there is a gap between two variables, this may indicate that the lower offset variable could be an array or an uninitialized struct. Understanding how Ghidra assigns variables on the stack aided in calculating the sizes of the variables.

The full behavioral tests can be seen in Appendix E.

7.2 Functional Tests

The functional tests showed that tracking the sources to sinks is a viable method for detecting overflows caused by vulnerable sinks. Comparing the expected and observed results, the buffer overflow detection method was successful for 95% of the test cases. Table 7.1 shows a summary of the test results. Full test results are in Appendix H. In most test cases the detection method was able to accurately estimate the allocated sizes and fill sizes of the sources. The primary causes of deviations in the size estimate were due to accounting for or not accounting for a null byte. The two cases that resulted in errors were due to the incorrect identification of a null byte from a good implementation of a sink and the inability to identify a source due to pointer usage.

Table 7.1. Summary of the results from the buffer overflow detection method

Test Case	Expected results	Actual results	TP	FP	FN	F-Score
1	1W	1W	1	0	0	1
2	1W	3W	1	2	0	.5
3	1W	1W	1	0	0	1
4	2C	2C	2	0	0	1
5	1W	1W	1	0	0	1
6	0W	0W	0	0	0	N/A*
7	0W	0W	0	0	0	N/A*
8	0W	0W	0	0	0	N/A*
9	1W	1W	1	0	0	1
10	1W	2W**	1	1	0	.67
11	1W	1W	1	0	0	1
12	0W	0W	0	0	0	N/A*
13	0W	0W	0	0	0	N/A*
14	0W	0W	0	0	0	N/A*
15	1W	1W	1	0	0	1
16	1W	1W	1	0	0	1
17	1W	1W	1	0	0	1
18	0W	0W	0	0	0	N/A*
19	0W	0W	0	0	0	N/A*

Table 7.1. Summary of the results from the buffer overflow detection method

Test Case	Expected results	Actual results	TP	FP	FN	F-Score
20	0W	0W	0	0	0	N/A*
21	1W, 2C	1W, 2C	3	0	0	1
22	1W, 2C	1W, 2C	3	0	0	1
23	1W, 1C	1W, 1C	2	0	0	1
24	1W, 1C	1W, 1C	2	0	0	1
25	1W, 1C	1W, 1C	2	0	0	1
26	1W, 1C	1W ,1C	2	0	0	1
27	1W, 1C	1W, 1C	2	0	0	1
28	1W	1W	1	0	0	1
29	1W	1W	1	0	0	1
30	1W	1W	1	0	0	1
31	0W	0W	0	0	0	N/A*
32	0W	0W	0	0	0	N/A*
33	0W	0W	0	0	0	N/A*
34	1W, 1C	1W, 1C	2	0	0	1
35	1W, 2C	1W, 2C	3	0	0	1
36	1W, 1C	1W, 1C	2	0	0	1
37	1W, 1C	1W, 1C	2	0	0	1
38	1W, 1C	1W, 1C	2	0	0	1
39	1W, 1C	1W, 1C	2	0	0	1
40	1W, 1C	1W, 1C	2	0	0	1
Total	27W, 19C	30W, 19C	46	3	0	.97

W = Warning message

C = Caution message

* = F-score calculation resulted in all zeros

** = Identified the wrong source

Decompile: good1 - (Test02)

```
1
2  void good1(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    undefined local_68 [60];
8    undefined4 local_2c;
9    wchar_t *local_28;
10   long local_10;
11
12   local_10 = *(long *)(in_FS_OFFSET + 0x28);
13   iVar1 = staticReturnsFalse();
14   if (iVar1 == 0) {
15     local_28 = L"0123456789abcdef0123456789abcde";
16     printWLine(L"0123456789abcdef0123456789abcde");
17     memmove(local_68,L"0123456789abcdef0123456789abcde",0x40);
18     local_2c = 0;
19     printWLine(local_68);
20     printWLine(local_28);
21   }
22   else {
23     printLine("Benign, fixed string");
24   }
25   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
26                     /* WARNING: Subroutine does not return */
27     __stack_chk_fail();
28   }
29   return;
30 }
31
```

Figure 7.1. Error in Test Case 2 caused by null byte

7.2.1 Review of the error rates

There were two cases that resulted in errors. First, Test Case 2 produced a warning for two good implementations of a sink. This is caused by the destination parameter not accounting for the null byte at the end of the char array. Figure 7.1 shows the Ghidra disassembly of the function that caused the error. We see that local_68 is the destination parameter and is allocated a space of 60 bytes. We also see that *memmove()* is writing 64 bytes (hex 0x40) to the destination parameter. At first glance this does seem like an overflow, however immediately below the *memmove()* call, a null byte is moved into local_2c. Local_2c is

a four-byte variable that resides immediately after local_68 (local_2c requires four bytes because local_68 is a UTF-32 wide char array). Since there is a direct reference to local_2c, Ghidra treats it as an independent variable, although this null byte value is part of the char array. Thus, with local_2c the actual size of the array is 64 bytes. This can be fixed by adding extra checks for the presence of a null bytes after a char array.

Test Case 10 produced two warnings when it should have produced one (see Listing 7.1). This is the result of the source location being misidentified as a pointer. Figure 7.2 shows the disassembly that caused the detection method to misidentify the source. Starting at the *memmove()* sink at the bottom of the figure, the source can be tracked up through code by following the last use of the operand. We see that the source is first at a pointer at offset -0x28. The value in offset -0x28 then came from the pointer at offset -0x38. Offset -0x38 is then loaded with the address of offset -0x48. Offset -0x48 is a pointer, not a source location. Since there is no instruction to load offset -0x48, the detection method did not find the actual value with which -0x48 is associated. The actual value of offset -0x48 is a pointer to offset -0x1d. This is the result of offset -0x30 pointing to offset -0x48 and then offset -0x1d is loaded into the location to which offset -0x30 points.

```
void ...CWE193_char_declare_memmove_32_bad()
{
    char * data;
    char * *dataPtr1 = &data;
    char * *dataPtr2 = &data;
    char dataBadBuffer[10];
    char dataGoodBuffer[10+1];
    {
        char * data = *dataPtr1;
        data = dataBadBuffer;
        data[0] = '\0';
        *dataPtr1 = data;
    }
    {
        char * data = *dataPtr2;
        {
            char source[10+1] = SRC_STRING;
            memmove(data, source, (strlen(source) + 1) * sizeof(char));
```

```
19              printLine(data);
20          }
21      }
22  }
```

Listing 7.1: Source code for Test Case 10

There is no simple solution for this case. This would require extensive analysis of pointer usage within functions to determine the actual source. Analysis like this would be better suited for dynamic analysis because the values that the pointers point to would need to be checked when they are used. This is needed because the values that the pointers point to can be different depending on the control flow that the program took.

Figure 7.2. Incorrect source identification due to pointer use. The left is the original disassembly. The right is the marked-up version.

7.3 Discussion

The tool focuses on function-to-function interactions, testing how sources pass through different functions to end up at a sink. Based on the testing results, tracking sources through parameter passing was demonstrated to be possible. Tracking sources inside a function illuminated some weaknesses. The test results showed that the tool needs a more robust means for tracking pointer usage to correctly identify sources. Other areas for improvements include improving the accuracy of the source size calculations, properly identifying data types in uninitialized structs, and determining sources from multiple register usage.

7.3.1 Source size calculation

The current method for calculating the size of the source relies on finding references to stack locations inside a given function. To determine a variable's size, the method calculates the unused space between the start of two variables. This method works well for initialized variables and cases where there is a reference to each of the variables. This method fails for direct references into arrays because Ghidra interprets direct references to stack locations as variables. Thus, calculating the space in between the variables will not work because Ghidra adds a new variable in the middle of an existing variable.

```
1  int main()
2  {
3      char string1[50];
4      char string2[50];
5
6      string1[13] = 'A';
7
8      for(int i = 0; i < 50; i++){
9          if(i == 13){
10             string2[i] = 'A';
11         }
12     }
13 }
```

Listing 7.2: Source calculation test

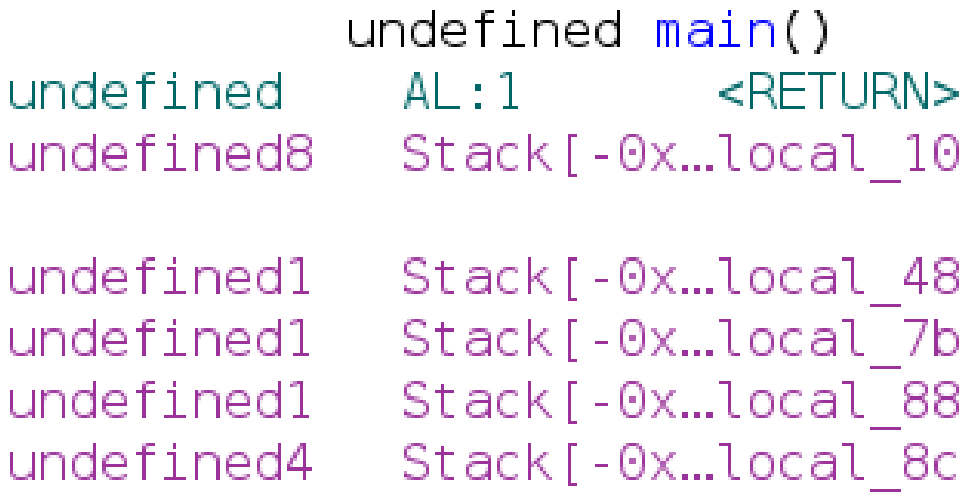


Figure 7.3. Disassembly of the stack layer for the source test

An example of direct referencing into a variable is seen in Listing 7.2 and Figure 7.3. Listing 7.2 shows a direct reference in the middle of a character array via static assignment within a loop. Figure 7.3 shows the stack layout that Ghidra produced from the binary code. Note that local_88 refers to string1 and local_48 refers to string2. For string1, we see a variable at location local_7b, which is the 13th index in the array. Using the current source size calculation method, the difference between local_88 and local_7b would result in the incorrect size of 13 for the variable. Without the source code, further investigation is needed to detect these cases.

7.3.2 Uninitialized structs

When structs are uninitialized, space on the stack frame is allocated for the whole struct. This results in the inability to differentiate individual data structures in a struct. To differentiate the different data structures, there needs to be a reference to each data structure's location in the code. Without references to the locations, calculating the distance between variables could result in calculating space that was meant for another data structure in the struct.

This problem does not have a good solution. One possible method would be to identify as many clues about the struct as possible. The information from functions like *sizeof()* make for good clues when calculating the length of a data type. This type of function sometimes

forces the compiler to statically assign the size value of the data type as an integer in the binary file. The statically assigned integer values allow for a direct means of understanding the space that a variable was allocated.

7.3.3 Multiple register tracking

The ODSS tool assumes that only one register is used to calculate a source's location. For the Juliet test cases used in this work, tracking one register was adequate for finding the sources. More complex programs use multiple registers to calculate a source's true location. This is seen in arrays of strings, when one register is used to point to a base location and another register is used to point to the offset into the particular string array. Tracking multiple registers can be implemented by searching for all the registers that are used in an instruction. Once the values for the registers are found, their values could be used to calculate the source's location.

7.3.4 Instruction set coverage

The ODSS tool focused on the instructions that appeared in the test sets. For example the SUB instruction could be handled similarly to the ADD instruction when calculating offsets. However, the SUB instruction did not appear in tests for tracking a source and, thus, the use of the SUB instruction for calculating offset remains untested. There are many other instructions that were also not handled; this is a subject area for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 8:
Conclusion

Overall, the ODSS script demonstrated that static analysis of binary programs could lead to the discovery of buffer overflow vulnerabilities. The bottom-up tracking process showed how to use the sinks and sources found in binary code to discover potential vulnerabilities. This method proved to be successful against a number of test cases in the Juliet test suite. The selected Juliet test cases were useful for testing variations of a sink's usage; however, those programs do not present the same challenges that exist in real world applications. One example is that the test cases relied on one register for indexing into the stack. Larger programs rely on multiple registers to index into a stack location, which is commonly the case for arrays. This chapter summarizes the motivation and results of this work and discusses potential enhancements to address the limitations of the current implementation.

8.1 Summary

The motivation for this work is to improve software assurance for the DoD by applying static analysis on binary code to determine the robustness of a program against buffer overflows. The research shows that it is possible to automate the discovery and testing of vulnerable sinks in a binary file for buffer overflows. The sink's parameters were tracked through the code to find the parameter's source locations. With the source locations, discovery of a source's usage was used to help determine the allocated size of the source and the largest string for that source. Once all the sources were calculated, the sources were compared against one another to determine if a buffer overflow was possible. This was implemented using the Ghidra API for searching, iterating, and referencing various aspects of the binary file. The final result was a Ghidra script written in Python to discover buffer overflows in the binary file being examined by Ghidra. Testing was conducted on vulnerable programs found inside of the Juliet test suite's stack-based overflow test cases. This method was successful for the Juliet test suite programs with an overall F-score of .97.

8.2 Future Work

This research aimed to demonstrate that the proposed method of tracking sinks and sources is feasible. Future work would address control flow inside functions, detecting overflows in variadic functions, testing with multiple types of compilers and compiler options, and providing additional information to the user. Work on these topics will allow our detection method to discover more overflows in the code and improve the range of programs that the ODSS script can produce useful results. Adding more vulnerable sinks to the test corpus would allow the script to discover more overflows. Using multiple types of compilers would allow the script to be used on more programs.

8.2.1 Control flow inside functions

The tool does not account for control flow inside of functions because basic source to sink tracking needed to be solved first. Understanding and tracking the control flow inside of a function is difficult, but an important feature of an effective buffer overflow detection tool. In many cases, a sink's parameters are not accessed with one register. Instead, they are accessed using values in several registers. These registers values can be different depending on the control flow of the program.

Control flow inside a function can be tracked in a manner similar to the function-to-function control flow, by building a tree to the source information. Instead of looking at references to the function calls, the tool would need to look at the different references to address locations caused by execution control instructions (e.g., JMP or Jcc instructions). To achieve this, each address needs to be checked for reference locations. Those reference locations become the nodes of the path tree and the search for sources is continued for each node. This method works for cases where there are direct references to a location. Cases like a JMP RAX would not work because the value for RAX is not known. Static analysis would not be the best options because it may not be possible to calculate register values. Thus, dynamic analysis would be better suited to handling this type of control flow.

8.2.2 Variadic functions

There are several variadic functions that can cause overflows that cannot be detected by the current detection method. Variadic functions require a format specifier to know how many

parameters to use. This format specifier is the starting point for discovering possible buffer overflows. For strings, variadic functions rely on the null byte to indicate the end of the string to be read. To limit the number of characters to read, a format specifier can use the width sub-specifier to specify the number of characters to read.

To implement a detection method for these functions, the format specifier must be parsed. The value from the format specifier becomes the input amount for that parameter. Once the parameter's width sub-specifier is calculated, the search for the sources can begin. Without a sub-specifier, the method would mark the function as an overflow risk, because there is no mechanism to prevent the buffer from being overflowed. The variadic-aware method operates the same as the current detection method by searching the code from the sinks to find the sources. Once the sources have been found, their allocated sizes can be compared against their respective format specifier. If the specifier is larger than the allocated size of the parameter, then an overflow is possible.

8.2.3 Multiple types of compilers and compiler options

The reason for choosing a specific compiler with specific options for the thesis was to create a control case without a lot of uncontrolled variables. For the tool to be more generally applicable, it must be improved to work with a wide array of compilers and options. In the test cases, the compiler consistently produced offsets for local variables using RBP but this behavior may change for other compiler options and compilers. For example, a different version of GCC could change its use of RBP, or using the same GCC version with different compiler options could generate offsets for local variables based on RSP. Cases like these would change the ways some attributes of a variable are calculated. In other cases, compilers may choose to inline functions making it harder to find where sinks occur. To find inlined functions, instruction templates need to be created for each sink. Then the entire code needs to be searched to find matches to these templates.

8.2.4 Increase testing on instruction set

Due to time constraints, the ODSS script only works with the x86 instructions used in the test programs. To improve the tool, additional logic should be added to the script to test the other instructions in the x86 instruction set to better account for source tracking and calculation.

8.2.5 User Feedback

The tool gives feedback on the sink and source information that it was able to find. There are cases where the tool makes estimates for the source's size and location. When an estimate needs to be used, information pertaining to the estimate could be displayed to the user. This would allow the user to help verify if the estimate is correct or to identify the correct information to use. There are cases where the values will not be known until run time. If run time information is needed, a message can be displayed to the user. Such messages would let the user know which sinks will need to be looked at via dynamic analysis.

APPENDIX A:
Implement a Ghidra Script

Ghidra has a well-developed API for performing various tasks on the disassembled information. The API allows users to write their own scripts to search, edit, or comment on disassembled information. The scripts can be written in Java or Python to utilize the API. This appendix describes how to implement a Ghidra script and run the script on a binary file.

First, the script manager needs to be opened. This can be done by opening a binary file and clicking on Window, then scrolling down to the script manager icon. The script manager icon can also be accessed from the top tool bar shown in Figure A.1.

Once opened the script manager's menu will appear. On the left side of the script manager is a list of the script folders. These folders are not file locations; instead they are categories into which the scripts belong. Categorization is based on the first few comments in the script file. At the top right there are two main buttons of interest. The first is the create script button. This will allow the user to select a Java or Python file to begin writing a script. It will also ask for the directory the file should be stored in. This will then open a text editor, allowing the user to begin writing code. The second button is the directory list. This allows the user to add directories for Ghidra to use when looking for script files. This is seen in Figure A.2.

When a script is selected, the file can be viewed and edited by clicking on the basic editor button. Inside the script, the top few comments determine what Ghidra will display for the script and how it is categorized. Comments starting with "@category:" determine where the file is stored in the script folders. For example, if "@category" is "@Bufferoverflow Detection", the script will be stored in the "Bufferoverflow Detection" folder.

To run a scripts, the script must be selected. Then, the green run script button will appear, which if selected, will run the script. This is seen in Figure A.3.

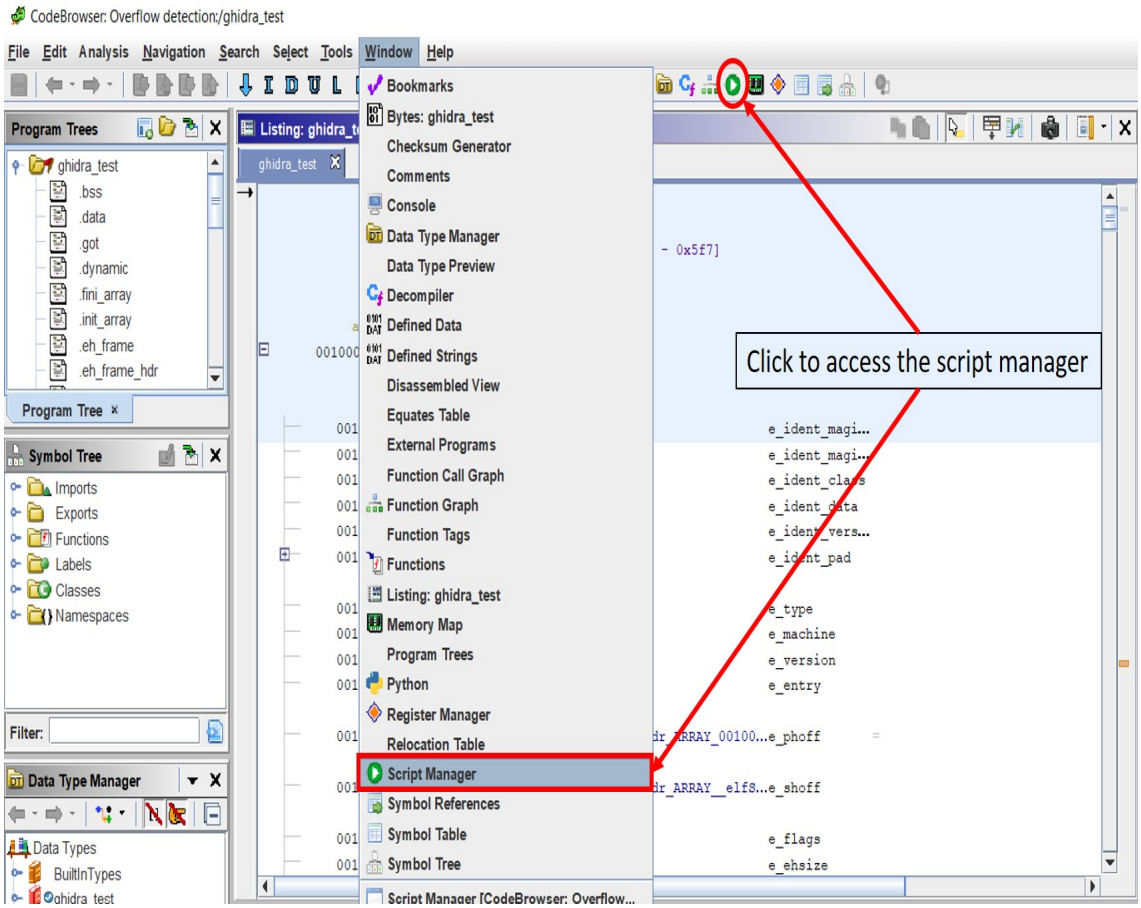


Figure A.1. Ghidra menu for opening the script menu

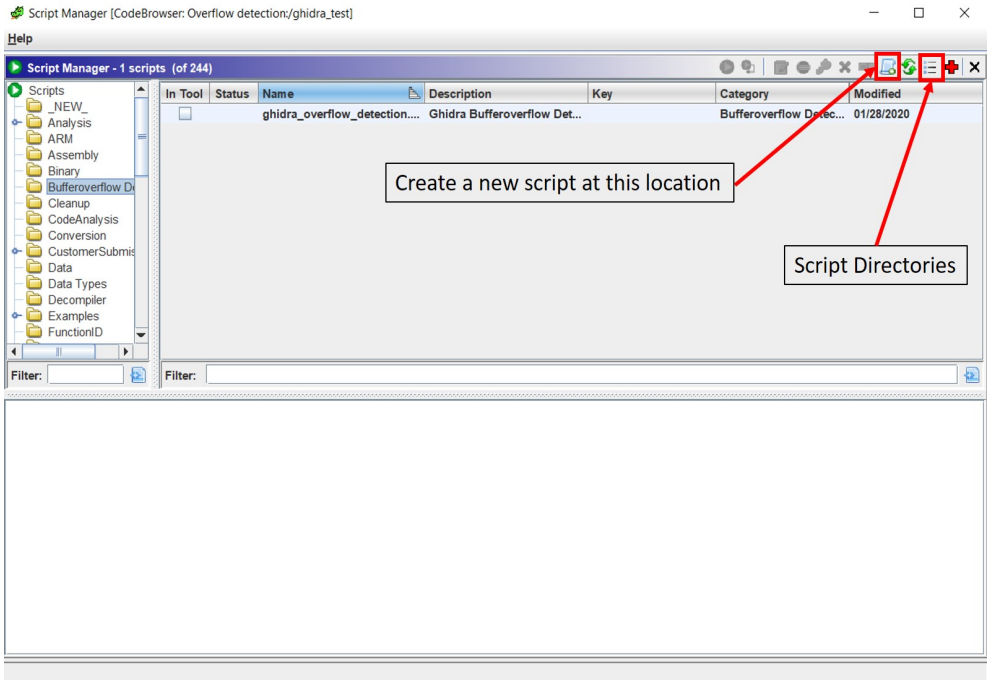


Figure A.2. Ghidra script manager window

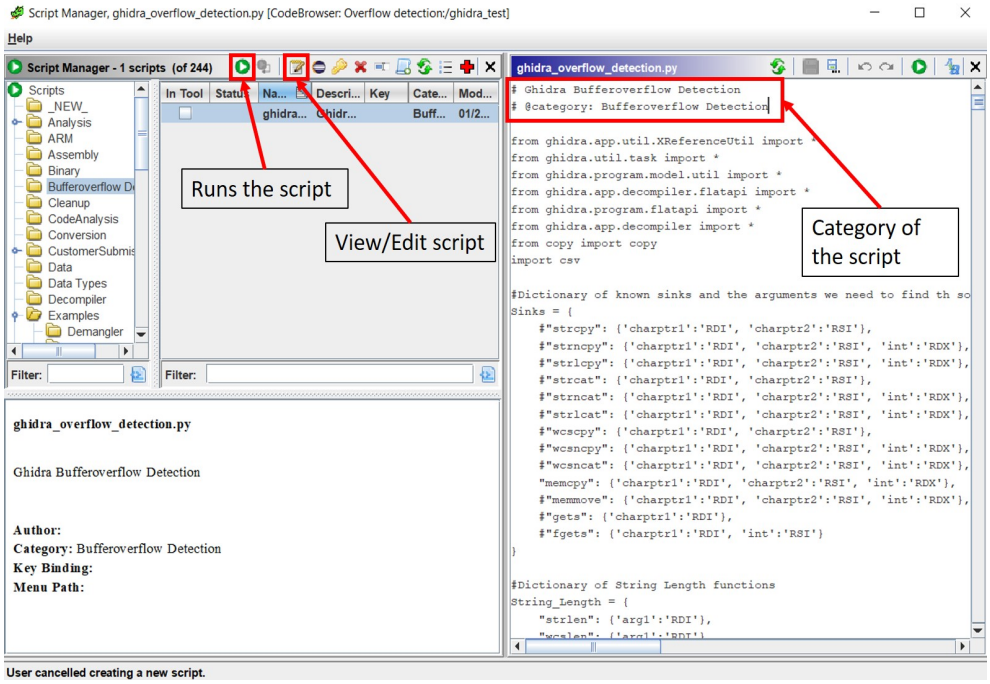


Figure A.3. Ghidra script manager for view, editing, and running scripts

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Test Case Flow Variant

This appendix describes the flow variants implemented in the Juliet test cases used for this work. Table B.1 was taken from the Juliet test suite [3].

The grouping of the flow variants are as follows:

- Flow variants 1-18: Uses conditional to control the flow of the program.
- Flow variants 21 & 22: Uses conditional to control the flow of the program and passes source information between functions.
- Flow variants 31 & 32: Uses pointers to point to the source values.
- Flow variant 34: Uses a union for the source parameter.
- Flow variants 41, 42, 44, & 45: Passes source information between functions.
- Flow variants 51-54: Passes source values between files.
- Flow variants 61-68 Pass values to different functions, but pass different types of information.

Table B.1 summarizes the flow variants.

Flow Variant	Flow Type	Description	C	C++
01	None	Baseline – Simplest form of the flaw	X	X
02	Control	if(1) and if(0)	X	X
03	Control	if(5==5) and if(5!=5)	X	X
04	Control	if(STATIC_CONST_TRUE) and if(STATIC_CONST_FALSE)	X	X
05	Control	if(staticTrue) and if(staticFalse)	X	X
06	Control	if(STATIC_CONST_FIVE==5) and if(STATIC_CONST_FIVE!=5)	X	X
07	Control	if(staticFive==5) and if(staticFive!=5)	X	X

Flow Variant	Flow Type	Description	C	C++
08	Control	if(staticReturnsTrue()) and if(staticReturnsFalse())	X	X
09	Control	if(GLOBAL_CONST_TRUE) and if(GLOBAL_CONST_FALSE)	X	X
10	Control	if(globalTrue) and if(globalFalse)	X	X
11	Control	if(globalReturnsTrue()) and if(globalReturnsFalse())	X	X
12	Control	if(globalReturnsTrueOrFalse())	X	X
13	Control	if(GLOBAL_CONST_FIVE==5) and if(GLOBAL_CONST_FIVE!=5)	X	X
14	Control	if(globalFive==5) and if(globalFive!=5)	X	X
15	Control	switch(6) and switch(7)	X	X
16	Control	while(1)	X	X
17	Control	for loops	X	X
18	Control	goto statements	X	X
21	Control	Flow controlled by value of a static global variable. All functions contained in one file.	X	X
22	Control	Flow controlled by value of a global variable. Sink functions are in a separate file from sources.	X	X
31	Data	Data flow using a copy of data within the same function	X	X
32	Data	Data flow using two pointers to the same value within the same function	X	X
33	Data	Use of a C++ reference to data within the same function	*	X
34	Data	Use of a union containing two methods of accessing the same data (within the same function)	X	X
41	Data	Data passed as an argument from one function to another in the same source file	X	X

Flow Variant	Flow Type	Description	C	C++
42	Data	Data returned from one function to another in the same source file	X	X
43	Data	Data flows using a C++ reference from one function to another in the same source file	*	X
44	Control/ Data	Data passed as an argument from one function to a function in the same source file called via a function pointer	X	X
45	Data	Data passed as a static global variable from one function to another in the same source file	X	X
51	Data	Data passed as an argument from one function to another in different source files	X	X
52	Data	Data passed as an argument from one function to another to another in three different source files	X	X
53	Data	Data passed as an argument from one function through two others to a fourth; all four functions are in different source files	X	X
54	Data	Data passed as an argument from one function through three others to a fifth; all five functions are in different source files	X	X
61	Data	Data returned from one function to another in different source files	X	X
62	Data	Data flows using a C++ reference from one function to another in different source files	*	X
63	Data	Pointer to data passed from one function to another in different source files	X	X
64	Data	void pointer to data passed from one function to another in different source files	X	X

Flow Variant	Flow Type	Description	C	C++
65	Control/ Data	Data passed as an argument from one function to a function in a different source file called via a function pointer	X	X
66	Data	Data passed in an array from one function to another in different source files	X	X
67	Data	Data passed in a struct from one function to another in different source files	X	X
68	Data	Data passed as a global variable in the "a" class from one function to another in different source files	X	X
72	Data	Data passed in a vector from one function to another in different source files	*	X
73	Data	Data passed in a linked list from one function to another in different source files	*	X
74	Data	Data passed in a hash map from one function to another in different source files	*	X
81	Data	Data passed in an argument to a virtual function called via a reference	*	X
82	Data	Data passed in an argument to a virtual function called via a pointer	*	X
83	Data	Data passed to a class constructor and destructor by declaring the class object on the stack	*	X
84	Data	Data passed to a class constructor and destructor by declaring the class object on the heap and deleting it after use	*	X

Table B.1. Flow Variants for the Juliet test suite [3]

APPENDIX C:
Compilation

The Juliet test programs are compiled using the GCC configurations listed in Appendix D. The programs contain macros to specify pieces of code that are or not part of the compilation process. These programs are compiled individually, which means each program requires a main function. These programs are also compiled with the GCC option -fno-builtin. This prevents the sinks from being inlined to the functions.

The header files std_testcases.h and std_testcases_io.h need to be in the same directory as the program files. These header files contain variables and macro definitions that are used in the test programs. The C program io.c is also needed in the compilation process. The program io.c handles the various print functions that are used in the test case programs.

The examples below use files from the Juliet test suite [3].

Single file compilation example:

```
gcc \DINCLUDEMAIN -fno-builtin io.c
CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_07.c
-o CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_07
```

In the cases of multiple files, each file will be included in the compilation process.
Multiple file compilation example:

```
gcc \DINCLUDEMAIN -fno-builtin io.c
CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_54a.c
CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_54b.c
CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_54c.c
CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_54d.c
CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_54e.c
-o CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_54
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D:
GCC Compiler Options

This Appendix lists the settings that were used by the GNU GCC compiler [32]. Multiple GCC commands were used to obtain the current settings.

Command used: gcc –version

```
Output:
gcc (Ubuntu 8.3.0-6ubuntu1) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Command used: gcc -Q –help=target

```
Output:
The following options are target specific:
  -m128bit-long-double         [enabled]
  -m16                         [disabled]
  -m32                         [disabled]
  -m3dnow                      [disabled]
  -m3dnowa                     [disabled]
  -m64                         [enabled]
  -m80387                      [enabled]
  -m8bit-idiv                  [disabled]
  -m96bit-long-double          [disabled]
  -mabi=                       sysv
  -mabm                        [disabled]
  -maccumulate-outgoing-args   [disabled]
  -maddress-mode=              long
  -madx                        [disabled]
  -maes                        [disabled]
  -malign-data=                compat
```

```
-malign-double               [disabled]
-malign-functions=           0
-malign-jumps=               0
-malign-loops=               0
-malign-stringops            [enabled]
-mandroid                    [disabled]
-march=                      x86-64
-masm=                       att
-mavx                        [disabled]
-mavx2                       [disabled]
-mavx256-split-unaligned-load  [enabled]
-mavx256-split-unaligned-store  [enabled]
-mavx5124fmaps               [disabled]
-mavx5124vnniw               [disabled]
-mavx512bitalg               [disabled]
-mavx512bw                   [disabled]
-mavx512cd                   [disabled]
-mavx512dq                   [disabled]
-mavx512er                   [disabled]
-mavx512f                    [disabled]
-mavx512ifma                 [disabled]
-mavx512pf                   [disabled]
-mavx512vbmi                 [disabled]
-mavx512vbmi2                [disabled]
-mavx512vl                   [disabled]
-mavx512vnni                 [disabled]
-mavx512vpopcntdq            [disabled]
-mbionic                     [disabled]
-mbmi                        [disabled]
-mbmi2                       [disabled]
-mbranch-cost=<0,5>          3
-mcall-ms2sysv-xlogues       [disabled]
-mcet-switch                 [disabled]
-mcld                        [disabled]
-mclflushopt                 [disabled]
```

```
-mclwb                          [disabled]
-mclzero                        [disabled]
-mcmodel=                       [default]
-mcpu=
-mcrc32                         [disabled]
-mcx16                          [disabled]
-mdispatch-scheduler            [disabled]
-mdump-tune-features            [disabled]
-mf16c                          [disabled]
-mfancy-math-387                [enabled]
-mfentry                        [disabled]
-mfma                           [disabled]
-mfma4                          [disabled]
-mforce-drap                    [disabled]
-mforce-indirect-call           [disabled]
-mfp-ret-in-387                 [enabled]
-mfpmath=                       sse
-mfsgsbase                      [disabled]
-mfunction-return=              keep
-mfused-madd
-mfxsr                          [enabled]
-mgeneral-regs-only            [disabled]
-mgfni                          [disabled]
-mglibc                         [enabled]
-mhard-float                    [enabled]
-mhle                           [disabled]
-miamcu                         [disabled]
-mieee-fp                       [enabled]
-mincoming-stack-boundary=      0
-mindirect-branch-register      [disabled]
-mindirect-branch=              keep
-minline-all-stringops          [disabled]
-minline-stringops-dynamically  [disabled]
-mintel-syntax
-mlarge-data-threshold=<number>  65536
```

```
-mlong-double-128           [disabled]
-mlong-double-64            [disabled]
-mlong-double-80            [enabled]
-mlwp                       [disabled]
-mlzcnt                     [disabled]
-mmemcpy-strategy=
-mmemset-strategy=
-mmitigate-rop              [disabled]
-mmmx                       [enabled]
-mmovbe                     [disabled]
-mmovdir64b                 [disabled]
-mmovdiri                   [disabled]
-mmpx                       [disabled]
-mms-bitfields              [disabled]
-mmusl                      [disabled]
-mmwaitx                    [disabled]
-mno-align-stringops        [disabled]
-mno-default                [disabled]
-mno-fancy-math-387         [disabled]
-mno-push-args              [disabled]
-mno-red-zone               [disabled]
-mno-sse4                   [enabled]
-mnop-mcount                [disabled]
-momit-leaf-frame-pointer   [disabled]
-mpc32                      [disabled]
-mpc64                      [disabled]
-mpc80                      [disabled]
-mpclmul                    [disabled]
-mpcommit                   [disabled]
-mpconfig                   [disabled]
-mpku                       [disabled]
-mpopcnt                    [disabled]
-mprefer-avx128
-mprefer-vector-width=      none
-mpreferred-stack-boundary= 0
```

```
-mprefetchwt1                [disabled]
-mprfchw                     [disabled]
-mpush-args                  [enabled]
-mrdpid                      [disabled]
-mrdrnd                      [disabled]
-mrdseed                     [disabled]
-mrecip                      [disabled]
-mrecip=
-mrecord-mcount              [disabled]
-mred-zone                   [enabled]
-mregparm=                   6
-mrtd                        [disabled]
-mrtm                        [disabled]
-msahf                       [disabled]
-msgx                        [disabled]
-msha                        [disabled]
-mshstk                      [disabled]
-mskip-rax-setup             [disabled]
-msoft-float                 [disabled]
-msse                        [enabled]
-msse2                       [enabled]
-msse2avx                    [disabled]
-msse3                       [disabled]
-msse4                       [disabled]
-msse4.1                     [disabled]
-msse4.2                     [disabled]
-msse4a                      [disabled]
-msse5
-msseregparm                 [disabled]
-mssse3                      [disabled]
-mstack-arg-probe            [disabled]
-mstack-protector-guard-offset=
-mstack-protector-guard-reg=
-mstack-protector-guard-symbol=
-mstack-protector-guard=     tls
```

```
-mstackrealign               [disabled]
-mstringop-strategy=         [default]
-mstv                        [enabled]
-mtbm                        [disabled]
-mtls-dialect=               gnu
-mtls-direct-seg-refs        [enabled]
-mtune-ctrl=
-mtune=                      generic
-muclibc                     [disabled]
-mvaes                       [disabled]
-mveclibabi=                 [default]
-mvect8-ret-in-mem           [disabled]
-mvpclmulqdq                 [disabled]
-mvzeroupper                 [enabled]
-mwbnoinvd                   [disabled]
-mx32                        [disabled]
-mxop                        [disabled]
-mxsave                      [disabled]
-mxsavec                     [disabled]
-mxsaveopt                   [disabled]
-mxsaves                     [disabled]

Known assembler dialects (for use with the -masm= option):
  att intel

Known ABIs (for use with the -mabi= option):
  ms sysv

Known code models (for use with the -mcmodel= option):
  32 kernel large medium small

Valid arguments to -mfpmath=:
  387 387+sse 387,sse both sse sse+387 sse,387

Known indirect branch choices
```

```
  (for use with the -mindirect-branch=/-mfunction-return= options):
    keep thunk thunk-extern thunk-inline

  Known data alignment choices (for use with the -malign-data= option):
    abi cacheline compat

  Known vectorization library ABIs (for use with the -mveclibabi= option):
    acml svml

  Known address mode (for use with the -maddress-mode= option):
    long short

  Known preferred register vector length
  (to use with the -mprefer-vector-width= option)
    128 256 512 none

  Known stack protector guard
  (for use with the -mstack-protector-guard= option):
    global tls

  Valid arguments to -mstringop-strategy=:
    byte\_loop libcall
        loop rep\_4byte rep\_8byte rep\_byte unrolled\_loop vector\_loop

  Known TLS dialects (for use with the -mtls-dialect= option):
    gnu gnu2
```

Command used: gcc -Q –help=optimizers

```
Output:
The following options control optimizations:
  -O<number>
  -Ofast
  -Og
  -Os
```

```
-faggressive-loop-optimizations  [enabled]
-falign-functions            [disabled]
-falign-functions=           1
-falign-jumps                [disabled]
-falign-jumps=               1
-falign-labels               [disabled]
-falign-labels=              1
-falign-loops                [disabled]
-falign-loops=               1
-fassociative-math           [disabled]
-fasynchronous-unwind-tables  [enabled]
-fauto-inc-dec               [enabled]
-fbranch-count-reg           [disabled]
-fbranch-probabilities       [disabled]
-fbranch-target-load-optimize  [disabled]
-fbranch-target-load-optimize2  [disabled]
-fbtr-bb-exclusive           [disabled]
-fcaller-saves               [disabled]
-fcode-hoisting              [disabled]
-fcombine-stack-adjustments  [disabled]
-fcompare-elim               [disabled]
-fconserve-stack             [disabled]
-fcprop-registers            [disabled]
-fcrossjumping               [disabled]
-fcse-follow-jumps           [disabled]
-fcx-fortran-rules           [disabled]
-fcx-limited-range           [disabled]
-fdce                        [enabled]
-fdefer-pop                  [disabled]
-fdelayed-branch             [disabled]
-fdelete-dead-exceptions     [disabled]
-fdelete-null-pointer-checks  [enabled]
-fdevirtualize               [disabled]
-fdevirtualize-speculatively  [disabled]
-fdse                        [enabled]
```

```
-fearly-inlining             [enabled]
-fexceptions                 [disabled]
-fexpensive-optimizations    [disabled]
-ffast-math
-ffinite-math-only           [disabled]
-ffloat-store                [disabled]
-fforward-propagate          [disabled]
-ffp-contract=[off|on|fast]  fast
-ffp-int-builtin-inexact     [enabled]
-ffunction-cse               [enabled]
-fgcse                       [disabled]
-fgcse-after-reload          [disabled]
-fgcse-las                   [disabled]
-fgcse-lm                    [enabled]
-fgcse-sm                    [disabled]
-fgraphite                   [disabled]
-fgraphite-identity          [disabled]
-fguess-branch-probability   [disabled]
-fhandle-exceptions
-fhoist-adjacent-loads       [disabled]
-fif-conversion              [disabled]
-fif-conversion2             [disabled]
-findirect-inlining          [disabled]
-finline                     [enabled]
-finline-atomics             [enabled]
-finline-functions           [disabled]
-finline-functions-called-once  [disabled]
-finline-small-functions     [disabled]
-fipa-bit-cp                 [disabled]
-fipa-cp                     [disabled]
-fipa-cp-clone               [disabled]
-fipa-icf                    [disabled]
-fipa-icf-functions          [disabled]
-fipa-icf-variables          [disabled]
-fipa-profile                [disabled]
```

```
-fipa-pta                     [disabled]
-fipa-pure-const              [disabled]
-fipa-ra                      [disabled]
-fipa-reference               [disabled]
-fipa-sra                     [disabled]
-fipa-vrp                     [disabled]
-fira-algorithm=[CB|priority]  CB
-fira-hoist-pressure          [enabled]
-fira-loop-pressure           [disabled]
-fira-region=[one|all|mixed]  [default]
-fira-share-save-slots        [enabled]
-fira-share-spill-slots       [enabled]
-fisolate-erroneous-paths-attribute  [disabled]
-fisolate-erroneous-paths-dereference  [disabled]
-fivopts                      [enabled]
-fjump-tables                 [enabled]
-fkeep-gc-roots-live          [disabled]
-flifetime-dse                [enabled]
-flifetime-dse=<0,2>          2
-flimit-function-alignment    [disabled]
-flive-range-shrinkage        [disabled]
-floop-interchange            [disabled]
-floop-nest-optimize          [disabled]
-floop-parallelize-all        [disabled]
-floop-unroll-and-jam         [disabled]
-flra-remat                   [disabled]
-fmath-errno                  [enabled]
-fmodulo-sched                [disabled]
-fmodulo-sched-allow-regmoves  [disabled]
-fmove-loop-invariants        [disabled]
-fnon-call-exceptions         [disabled]
-fnothrow-opt                 [disabled]
-fomit-frame-pointer          [disabled]
-fopt-info                    [disabled]
-foptimize-sibling-calls      [disabled]
```

```
-foptimize-strlen            [disabled]
-fpack-struct                [disabled]
-fpack-struct=<number>
-fpartial-inlining           [disabled]
-fpatchable-function-entry=
-fpeel-loops                 [disabled]
-fpeephole                   [enabled]
-fpeephole2                  [disabled]
-fplt                        [enabled]
-fpredictive-commoning       [disabled]
-fprefetch-loop-arrays       [enabled]
-fprintf-return-value        [enabled]
-freciprocal-math            [disabled]
-freg-struct-return          [enabled]
-frename-registers           [enabled]
-freorder-blocks             [disabled]
-freorder-blocks-algorithm=[simple|stc]  simple
-freorder-blocks-and-partition  [disabled]
-freorder-functions          [disabled]
-frerun-cse-after-loop       [disabled]
-freschedule-modulo-scheduled-loops  [disabled]
-frounding-math              [disabled]
-frtti                       [enabled]
-fsched-critical-path-heuristic  [enabled]
-fsched-dep-count-heuristic  [enabled]
-fsched-group-heuristic      [enabled]
-fsched-interblock           [enabled]
-fsched-last-insn-heuristic  [enabled]
-fsched-pressure             [disabled]
-fsched-rank-heuristic       [enabled]
-fsched-spec                 [enabled]
-fsched-spec-insn-heuristic  [enabled]
-fsched-spec-load            [disabled]
-fsched-spec-load-dangerous  [disabled]
-fsched-stalled-insns        [disabled]
```

```
-fsched-stalled-insns-dep    [enabled]
-fsched-stalled-insns-dep=<number>
-fsched-stalled-insns=<number>
-fsched2-use-superblocks     [disabled]
-fschedule-fusion            [enabled]
-fschedule-insns             [disabled]
-fschedule-insns2            [disabled]
-fsection-anchors            [disabled]
-fsel-sched-pipelining       [disabled]
-fsel-sched-pipelining-outer-loops  [disabled]
-fsel-sched-reschedule-pipelined  [disabled]
-fselective-scheduling       [disabled]
-fselective-scheduling2      [disabled]
-fshort-enums                [enabled]
-fshort-wchar                [disabled]
-fshrink-wrap                [disabled]
-fshrink-wrap-separate       [enabled]
-fsignaling-nans             [disabled]
-fsigned-zeros               [enabled]
-fsimd-cost-model=[unlimited|dynamic|cheap]  unlimited
-fsingle-precision-constant  [disabled]
-fsplit-ivs-in-unroller      [enabled]
-fsplit-loops                [disabled]
-fsplit-paths                [disabled]
-fsplit-wide-types           [disabled]
-fssa-backprop               [enabled]
-fssa-phiopt                 [disabled]
-fstack-check=[no|generic|specific]
-fstack-clash-protection     [disabled]
-fstack-protector            [disabled]
-fstack-protector-all        [disabled]
-fstack-protector-explicit   [disabled]
-fstack-protector-strong     [disabled]
-fstack-reuse=[all|named\_vars|none]  all
-fstdarg-opt                 [enabled]
```

```
-fstore-merging              [disabled]
-fstrict-aliasing            [disabled]
-fstrict-enums               [disabled]
-fstrict-volatile-bitfields  [enabled]
-fthread-jumps               [disabled]
-fno-threadsafe-statics      [enabled]
-ftracer                     [disabled]
-ftrapping-math              [enabled]
-ftrapv                      [disabled]
-ftree-bit-ccp               [disabled]
-ftree-builtin-call-dce      [disabled]
-ftree-ccp                   [disabled]
-ftree-ch                    [disabled]
-ftree-coalesce-vars         [disabled]
-ftree-copy-prop             [disabled]
-ftree-cselim                [enabled]
-ftree-dce                   [disabled]
-ftree-dominator-opts        [disabled]
-ftree-dse                   [disabled]
-ftree-forwprop              [enabled]
-ftree-fre                   [disabled]
-ftree-loop-distribute-patterns  [disabled]
-ftree-loop-distribution     [disabled]
-ftree-loop-if-convert       [enabled]
-ftree-loop-im               [enabled]
-ftree-loop-ivcanon          [enabled]
-ftree-loop-optimize         [enabled]
-ftree-loop-vectorize        [disabled]
-ftree-lrs                   [disabled]
-ftree-parallelize-loops=<number>  1
-ftree-partial-pre           [disabled]
-ftree-phiprop               [enabled]
-ftree-pre                   [disabled]
-ftree-pta                   [disabled]
-ftree-reassoc               [enabled]
```

```
-ftree-scev-cprop            [enabled]
-ftree-sink                  [disabled]
-ftree-slp-vectorize         [disabled]
-ftree-slsr                  [disabled]
-ftree-sra                   [disabled]
-ftree-switch-conversion     [disabled]
-ftree-tail-merge            [disabled]
-ftree-ter                   [disabled]
-ftree-vectorize
-ftree-vrp                   [disabled]
-funconstrained-commons      [disabled]
-funroll-all-loops           [disabled]
-funroll-loops               [disabled]
-funsafe-math-optimizations  [disabled]
-funswitch-loops             [disabled]
-funwind-tables              [enabled]
-fvar-tracking               [enabled]
-fvar-tracking-assignments   [enabled]
-fvar-tracking-assignments-toggle  [disabled]
-fvar-tracking-uninit        [disabled]
-fvariable-expansion-in-unroller  [disabled]
Specifies the cost model for vectorization.
-fvect-cost-model=[unlimited|dynamic|cheap]  [default]
-fvpt                        [disabled]
-fweb                        [enabled]
-fwrapv                      [disabled]
-fwrapv-pointer              [disabled]
```

APPENDIX E:
Behavioral Test

To utilize Ghidra's API, it was necessary to understand how Ghidra handled parameters that were passed as registers and how Ghidra identified local variables. A large part of this thesis revolved around the ability to track variables back to their source. This was tricky with 64-bit operating systems because functions pass the first six parameters in registers and the remaining parameters are passed on the stack. Listing E.1 is the test program we used to see how Ghidra handled the parameters as they were passed to different types of functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include <limits.h>

struct struct_test
{
   char test1[50];
   char test2[100];
   char test3[50];
   char test4[20];
   int test5;
};

int global_test;
char global_test2;
void test1_no_par();
void test2_one_par(int par1);
void test3_two_par(int par1, int par2);
void test4_two_par_ptr(char *par1, char *par2);
```

```
26 void test5_seven_par(int par1, int par2, char *par3, char *par4, char *
       par5, char *par6, char *par7);
27 void test6_one_par_call(int par1);
28 void test7_two_par_call(char *par1, int par2);
29 void test8_seven_par_call(int par1, int par2, char *par3, char *par4,
       char *par5, char *par6, char *par7);
30 void test9_struct_init(struct struct_test *);
31 void test10_struct_uninit(struct struct_test *);
32 void test11_struct_by_value(struct struct_test struct1);
33 void test12_casting(char charpar, int ipar, float fpar);
34
35 int main()
36 {
37     int ipar1 = 10;
38   int ipar2 = 20;
39   int ipar3 = 30;
40   float fpar1 = 10.1;
41   char charpar1 = 'a';
42   char cpar1[] = {"Hello World"};
43   char cpar2[50] = {"abcdefghijklmnopqrstuvwxyz"};
44   char cpar3[] = {"123456789"};
45   char cpar4[] = {"test test test"};
46   char cpar5[] = {"as;lfjaslkdfnalkfhaslk"};
47   struct struct_test struct1 = {"", "", "", "", 0};
48   struct struct_test struct2;
49   struct struct_test struct3 = {"TEST TEST", "TEST TEST", "TEST TEST", "
       TEST TEST", 4};
50
51   test1_no_par();
52   test2_one_par(ipar1);
53   test3_two_par(ipar1, ipar2);
54   test4_two_par_ptr(cpar1, cpar2);
55   test5_seven_par(ipar1, ipar2, cpar1, cpar2, cpar3, cpar4, cpar5);
56   test6_one_par_call(ipar3);
57   test7_two_par_call(cpar1, ipar2);
58   test8_seven_par_call(ipar1, ipar2, cpar1, cpar2, cpar3, cpar4, cpar5);
59   test9_struct_init(&struct1);
60   test10_struct_uninit(&struct2);
61   test11_struct_by_value(struct3);
62   test12_casting(charpar1, ipar1, fpar1);
```

```
63
64 }
65
66 void test1_no_par()
67 {
68   char cpar1[] = {"Hello test1"};
69   printf("%s\n", cpar1);
70 }
71
72 void test2_one_par(int par1)
73 {
74   int lvar = 2;
75   int total = 0;
76   total = lvar + par1;
77 }
78
79 void test3_two_par(int par1, int par2)
80 {
81   global_test = par1 + par2;
82 }
83
84 void test4_two_par_ptr(char *par1, char *par2)
85 {
86   global_test2 = par1[0];
87   global_test2 = par2[0];
88 }
89
90 void test5_seven_par(int par1, int par2, char *par3, char *par4, char *
       par5, char *par6, char *par7)
91 {
92   int lvar1 = par1;
93   int lvar2 = par2;
94   char lvar3 = par3[0];
95   char lvar4 = par4[0];
96   char lvar5 = par5[0];
97   char lvar6 = par6[0];
98   char lvar7 = par7[0];
99 }
100
101 void test6_one_par_call(int par1)
```

```
102 {
103   int total = par1 + 5;
104   printf("%i\n", total);
105 }
106
107 void test7_two_par_call(char *par1, int par2)
108 {
109   char lvar1 = par1[0];
110         int lvar2 = par2;
111   printf("%c %i\n", lvar1, lvar2);
112 }
113
114 void test8_seven_par_call(int par1, int par2, char *par3, char *par4,
      char *par5, char *par6, char *par7)
115 {
116   int lvar1 = par1;
117   int lvar2 = par2;
118   char lvar3 = par3[0];
119   char lvar4 = par4[0];
120   char lvar5 = par5[0];
121   char lvar6 = par6[0];
122   char lvar7 = par7[0];
123   printf("%i %i %s %s %s %s %s", lvar1, lvar2, par3, par4, par5, par6,
      par7);
124 }
125
126 void test9_struct_init(struct struct_test *struct1)
127 {
128   printf("%s", struct1->test1);
129   printf("%s", struct1->test2);
130   printf("%s", struct1->test3);
131   printf("%s", struct1->test4);
132   printf("%d", struct1->test5);
133   strcpy(struct1->test1, struct1->test2);
134 }
135
136 void test10_struct_uninit(struct struct_test *struct1)
137 {
138   printf("%s", struct1->test1);
139   printf("%s", struct1->test2);
```

```
140   printf("%s", struct1->test3);
141   printf("%s", struct1->test4);
142   printf("%d", struct1->test5);
143   strcpy(struct1->test4, struct1->test3);
144 }
145
146 void test11_struct_by_value(struct struct_test struct1)
147 {
148   printf("%s", struct1.test1);
149   printf("%s", struct1.test2);
150   printf("%s", struct1.test3);
151   printf("%s", struct1.test4);
152   printf("%d", struct1.test5);
153   strcpy(struct1.test1, struct1.test2);
154 }
155
156 void test12_casting(char charpar, int ipar, float fpar)
157 {
158   char local_str1[10];
159   char local_str2[20];
160   int local_int1 = (int)fpar;
161   int local_int2 = (int)charpar;
162   char local_char = (char)ipar;
163   float local_float = (float)ipar;
164   global_test2 = (int)fpar;
165   printf("%d", local_int1);
166   printf("%d", local_int2);
167   printf("%c", local_char);
168   printf("%f", local_float);
169   strcpy(local_str1, local_str2);
170   strncpy(local_str1, local_str2 ,(int)charpar);
171   strncpy(local_str1, local_str2 ,(int)fpar);
172
173 }
```

Listing E.1: Source Code for testing Ghidra's decomiler on assembly code

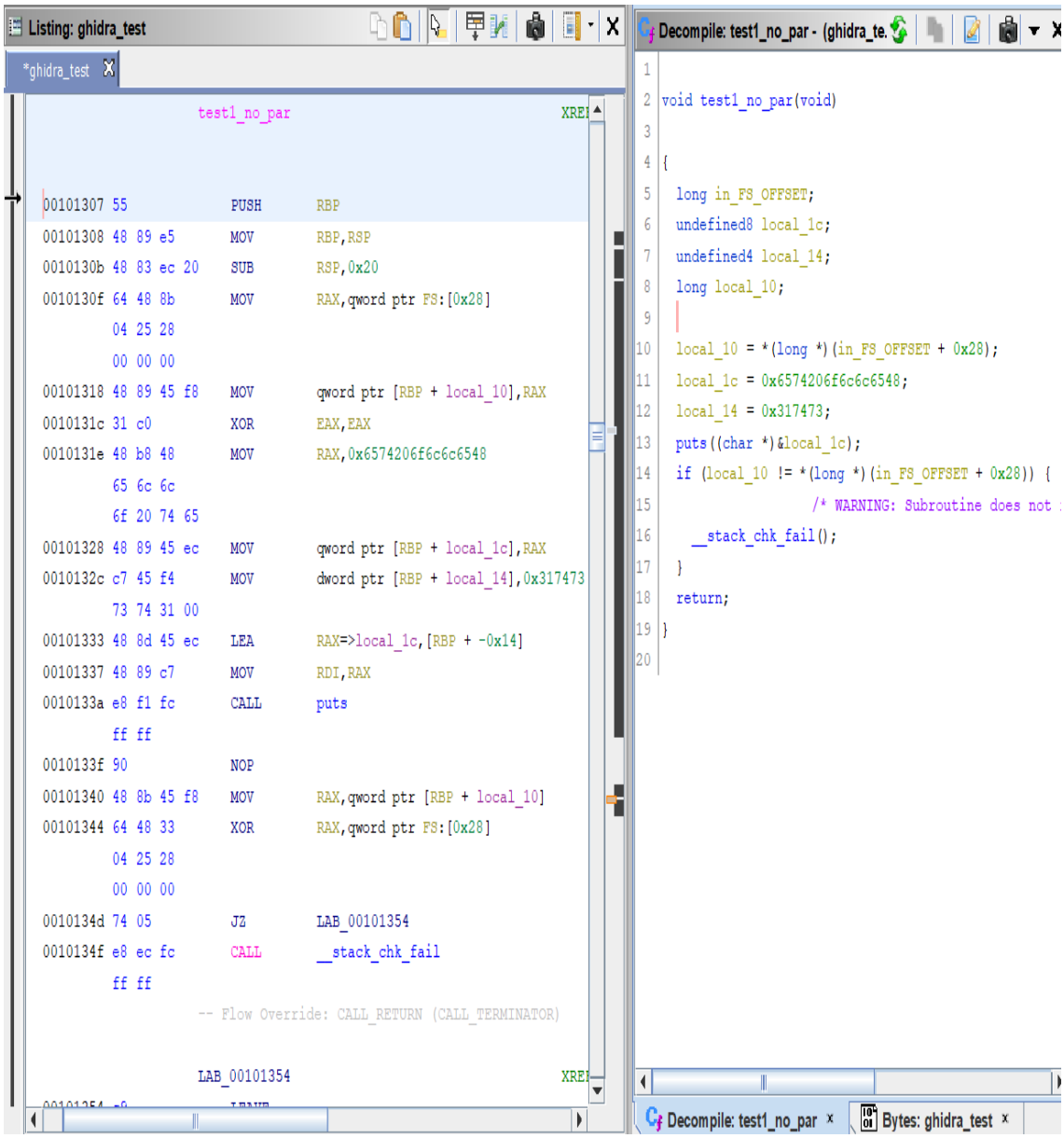


Figure E.1. Test 1: Function with no parameters

Test 1 was to observe the output from a function that took in no parameters. In Figure E.1 we see that Ghidra successfully identified that no parameters were passed to the function test1_no_par.

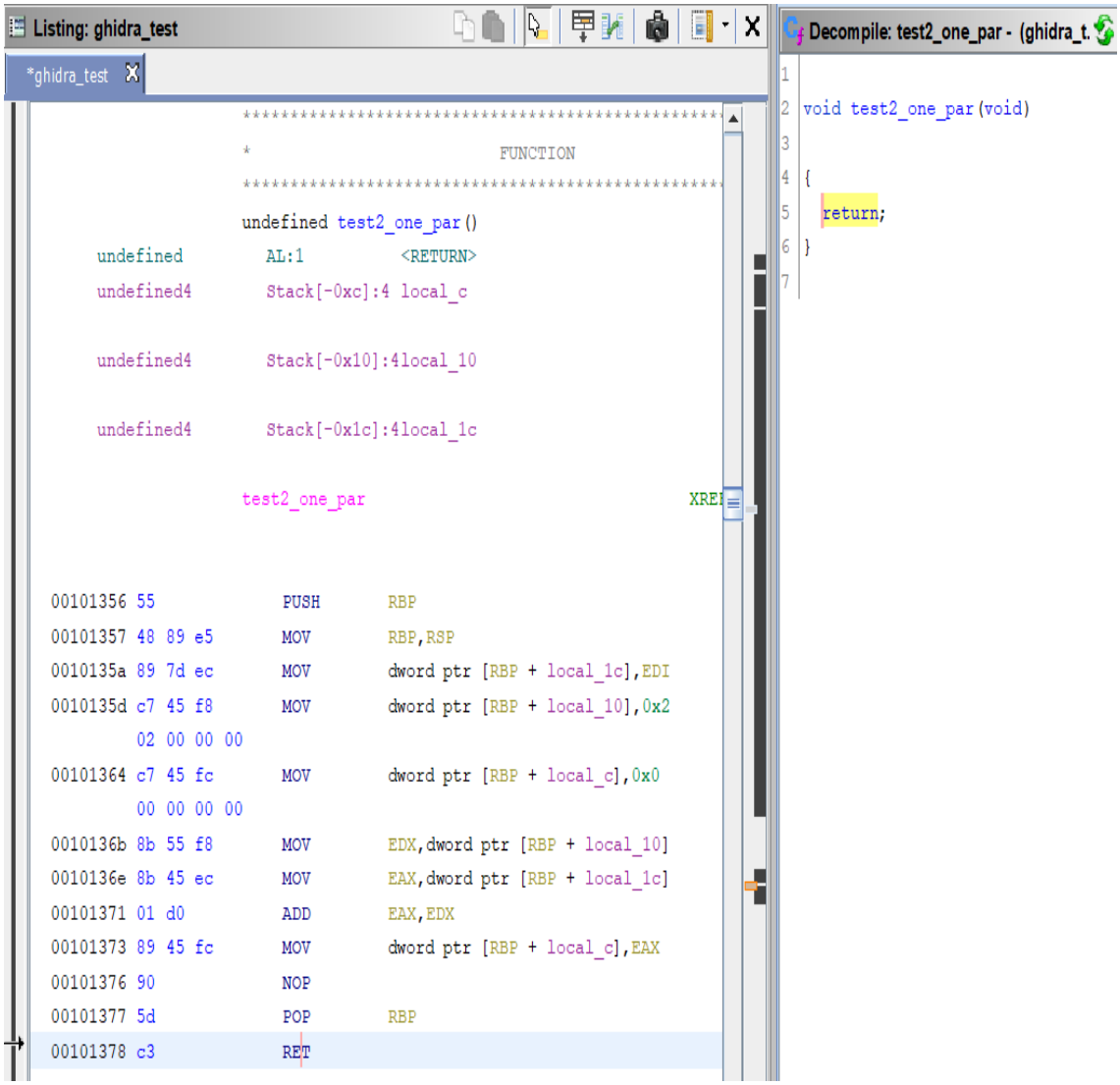


Figure E.2. Test 2: Function with one parameter

Test 2 passed one parameter to the function. This function performed a simple addition using the parameter and a local variable. This function ultimately did nothing because the sum was stored in a local variable and was not returned to the previous function. Figure E.2 shows two interesting aspects about Ghidra. First, Ghidra recognized the parameter as a local variable. This may be in part because the compiler stored the content of the registers as local variables; however, later we will see that other functions will label parameters differently. Second, the decompiler recognized that the function did nothing, so an empty function was the result of the decompilation.

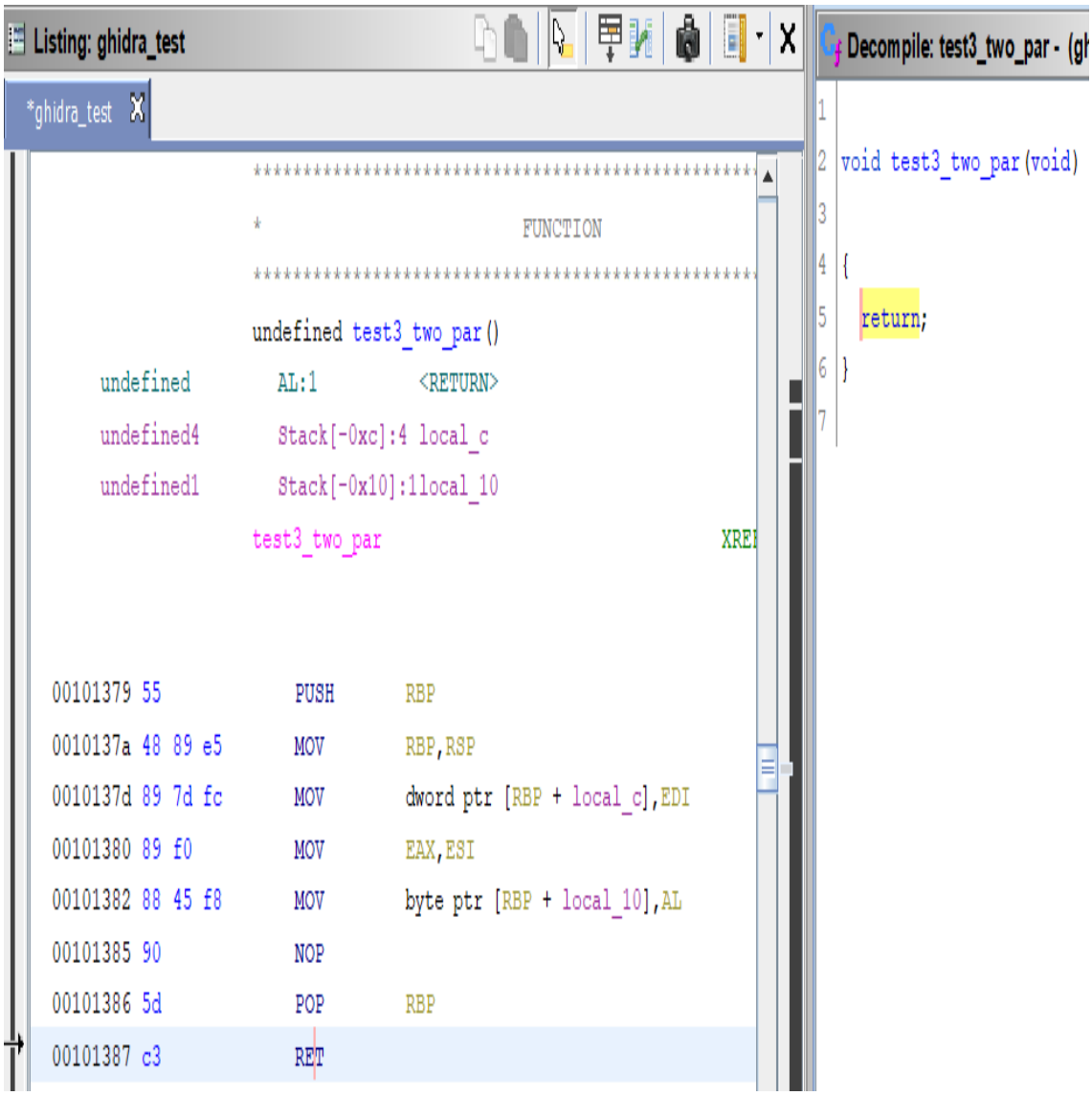


Figure E.3. Test 3: Function with two parameters

Test 3 passed two parameters to a function with no content. Figure E.3 shows that Ghidra's decompiler recognized that the function did nothing, and it did not list the two parameters that were passed to the function.

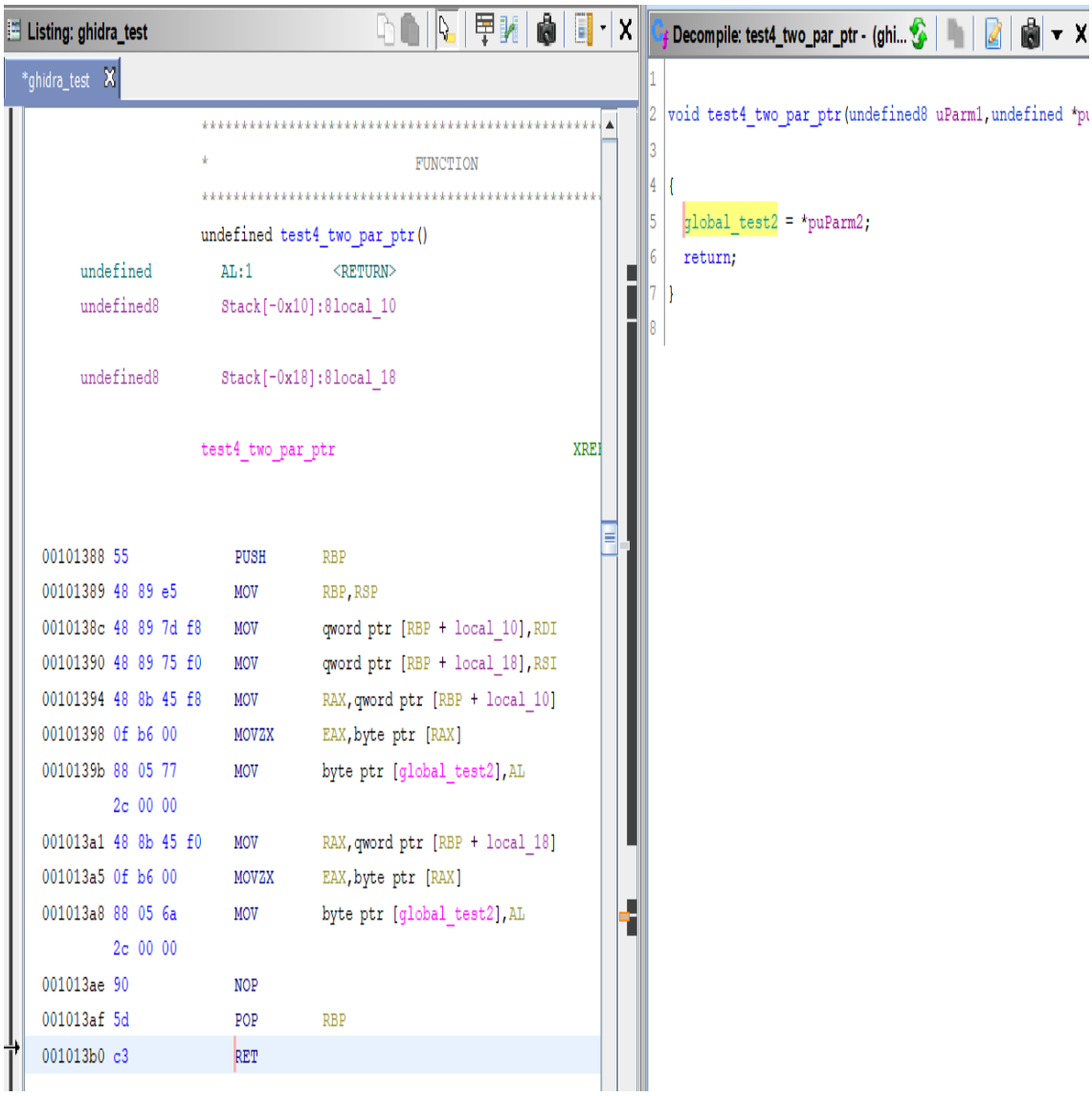


Figure E.4. Test 4: Function with two parameters passed as pointers

Test 4 passed two parameters as pointers to the function. This function wrote the content of the two parameters to the same global variable. Figure E.4 shows that the disassembler recognized the parameters as local variables, yet the decompiler recognized the parameters as parameters. We also notice that the decompiler recognized that the first assignment to the global variable was overwritten by the second assignment. So, the decompiler only displayed the second assignment in the decompiled view.

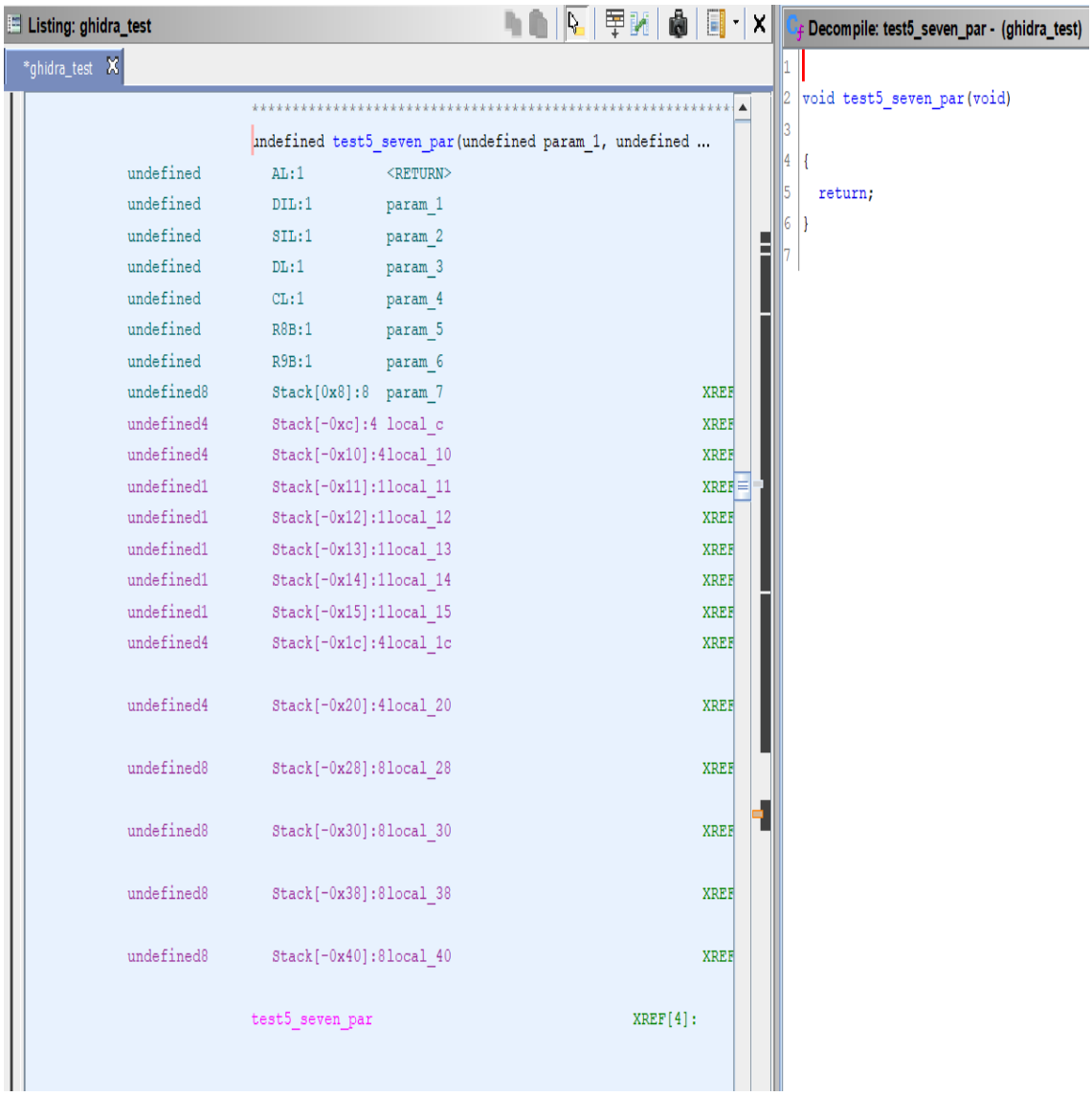


Figure E.5. Test 5: Function with seven parameters

Test 5 passed seven parameters to the function. This test was meant to observe how Ghidra would recognize the seventh parameter that was passed on the stack. The function itself assigned the parameters as local variables, which is the equivalent of doing nothing since nothing is returned. Figure E.5 shows that Ghidra recognized all the parameters as parameters, as seen as param_1 through param_7. We also notice that the decompiler recognized that the function did nothing and produced an empty function.

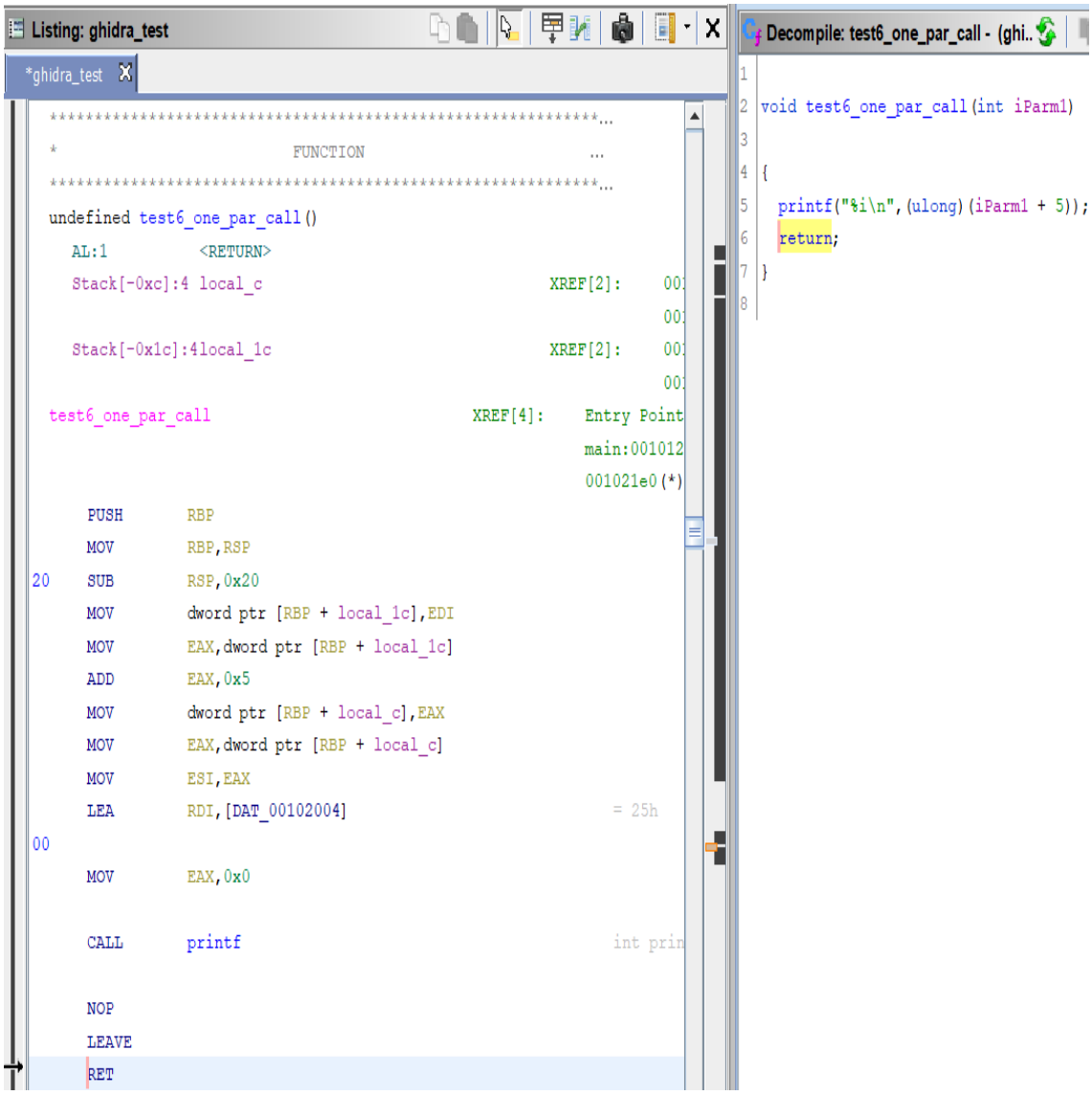


Figure E.6. Test 6: Function with one parameter and a function call

Test 6 passed one parameter to the function that computes a sum and makes one call to *printf()*. Our intent for this function was to see how Ghidra would recognize the parameter that was stored in the function before it was used to *printf()*. Figure E.6 shows that the decompiler summarized the sum part of the function and put the sum directly into the *printf()* statement. Also, we see that the decompiler recognized that the parameter was used versus a local variable.

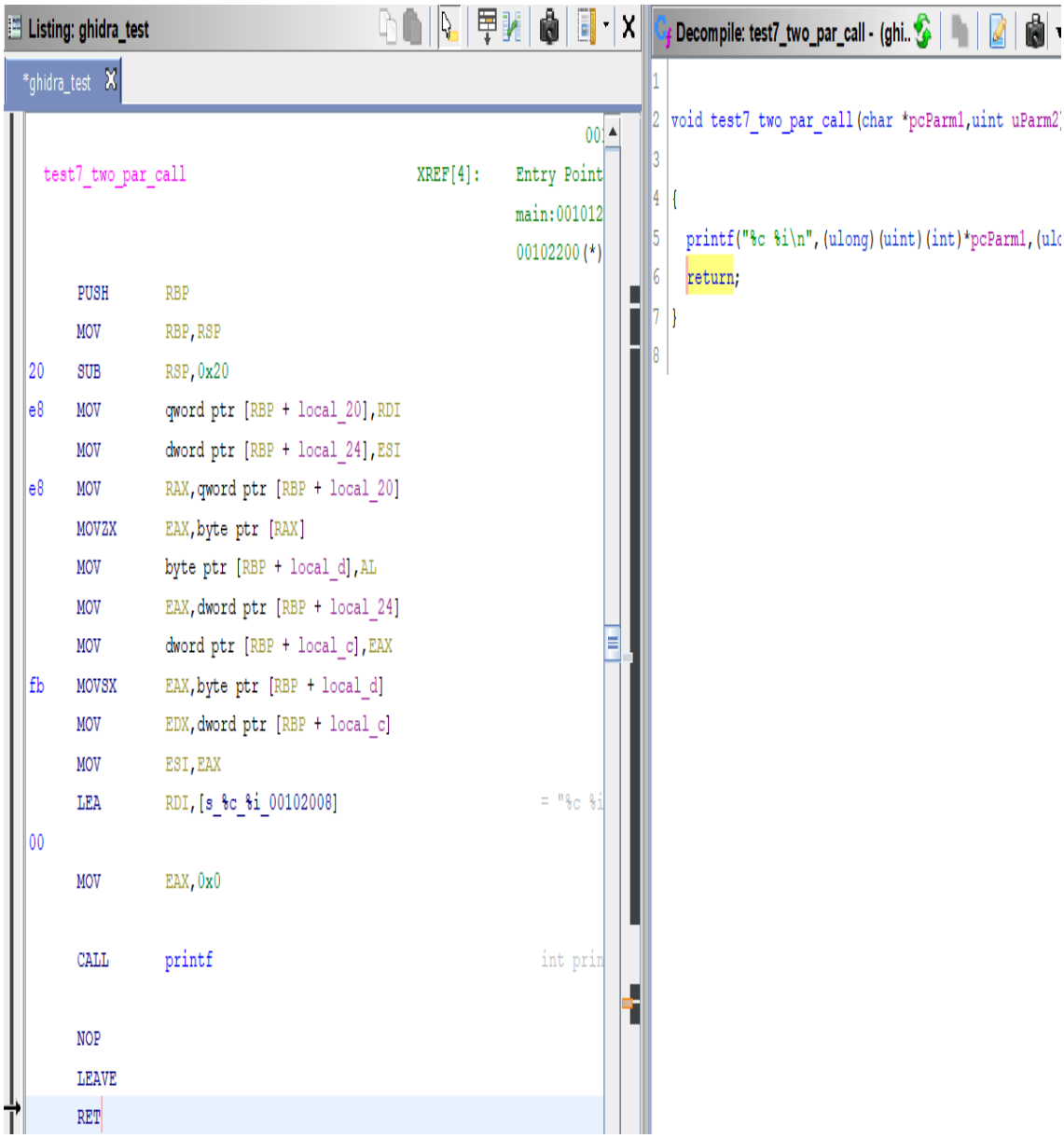


Figure E.7. Test 7: Function with two parameters and a function call

Test 7 passed two parameters to a function that calls *printf()* from local variables. As was the case for Test 6, Figure E.7 shows that the decompiler optimized the function by only passing the parameters into *printf()*. The assembly code shows that the parameters are moved into local variables before they are used in the *printf()* function.

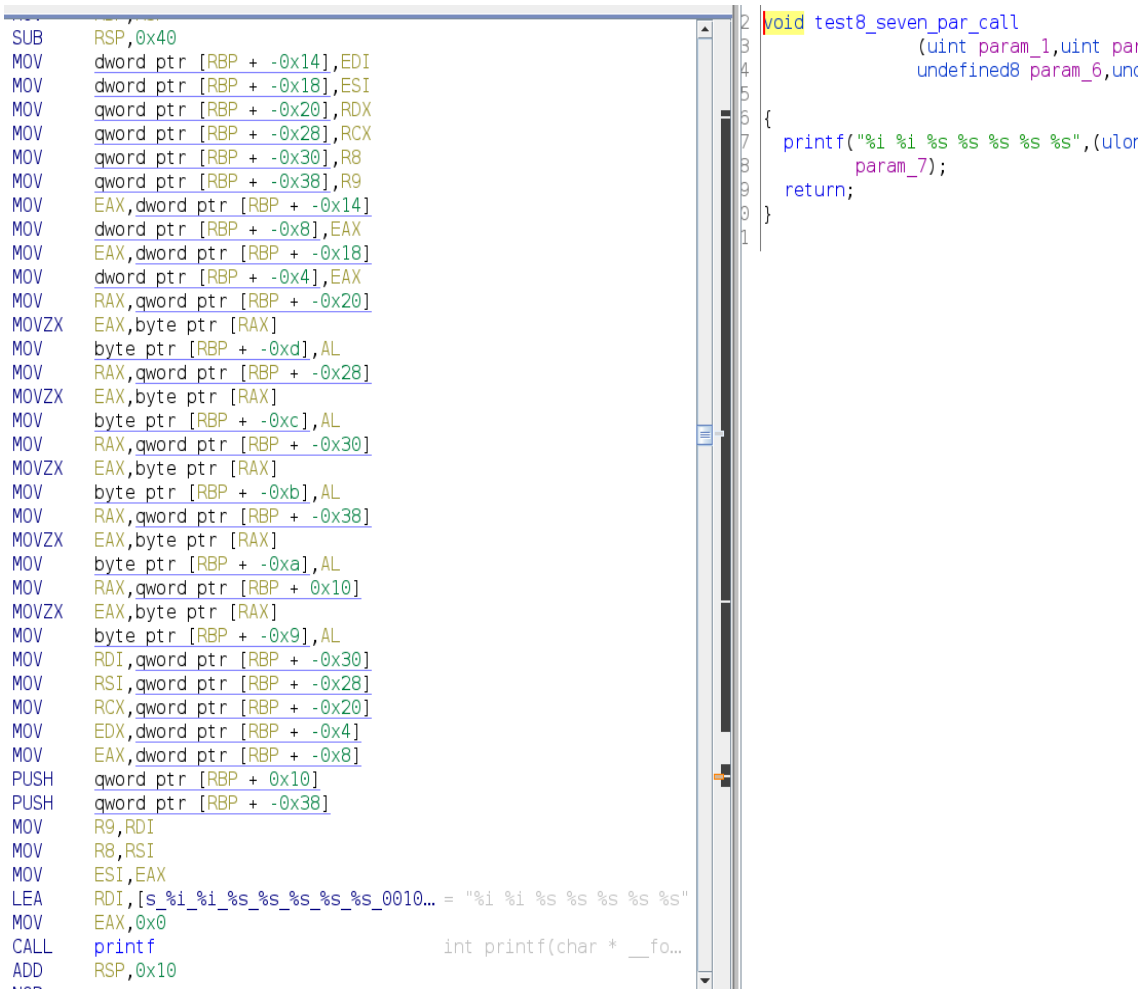
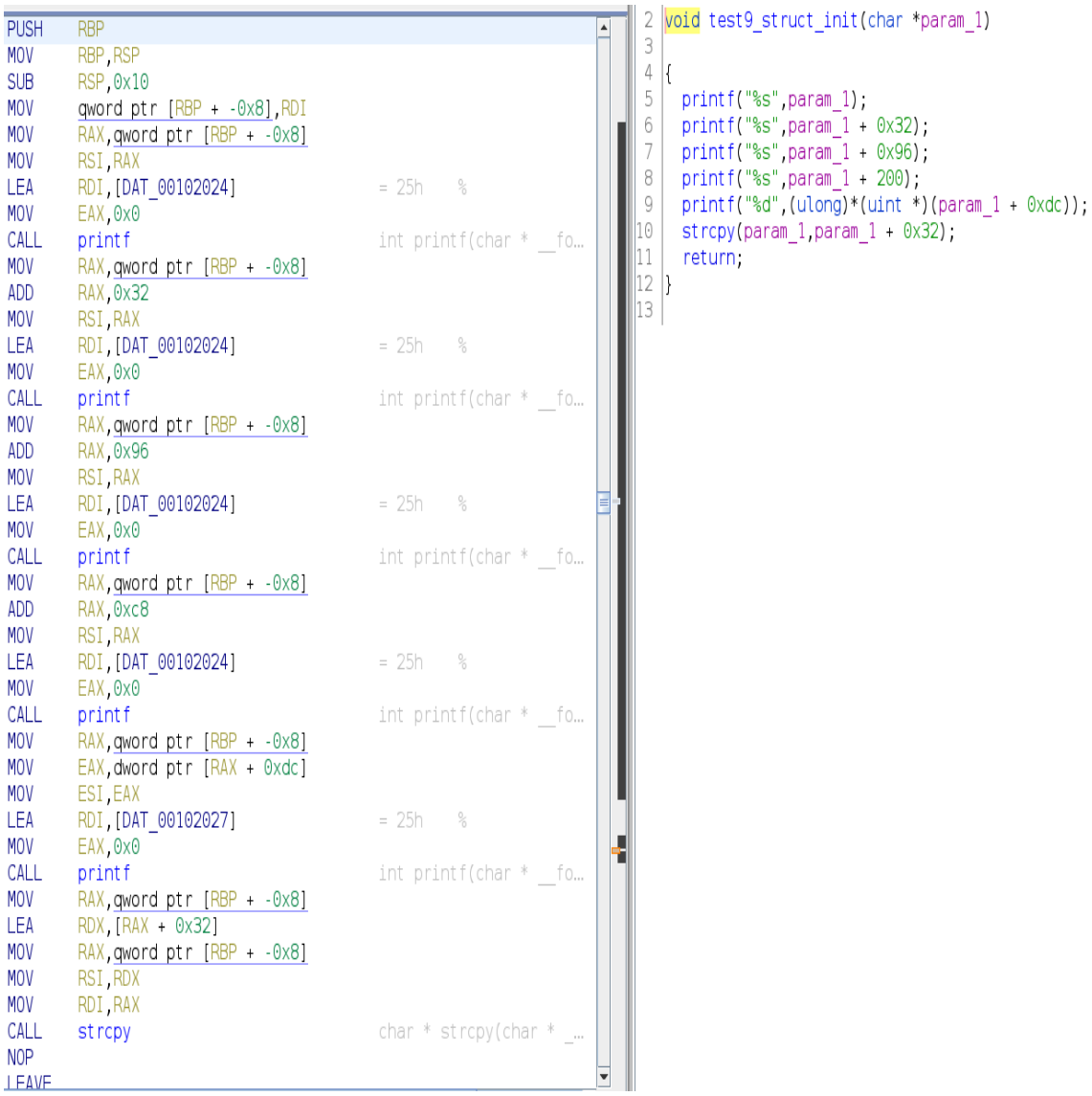


Figure E.8. Test 8: Function with seven parameters and a function call

Test 8 passed seven parameters to a function that calls *printf()* using only one of the parameters. Figure E.8 shows that Ghidra recognized the parameters as parameters in the disassembly code. At the bottom of the figure we see that param_1 and param_2 is used to pass *printf()* its values. In this case param_1 represents RDI and param_2 represents RSI. The decompiler recognized that only one parameter is used in the function. In this case the decompiler labeled the parameter as param_1 and passed that parameter into *printf()*. This gets a little convoluted because the assembly code shows param_2 as the variable going into *printf()* and the decompiler shows param_1 going into *printf()*. In the end, it is the same value going into the *printf()*, but Ghidra mixed up the symbols when comparing the assembly to the decompiled code.

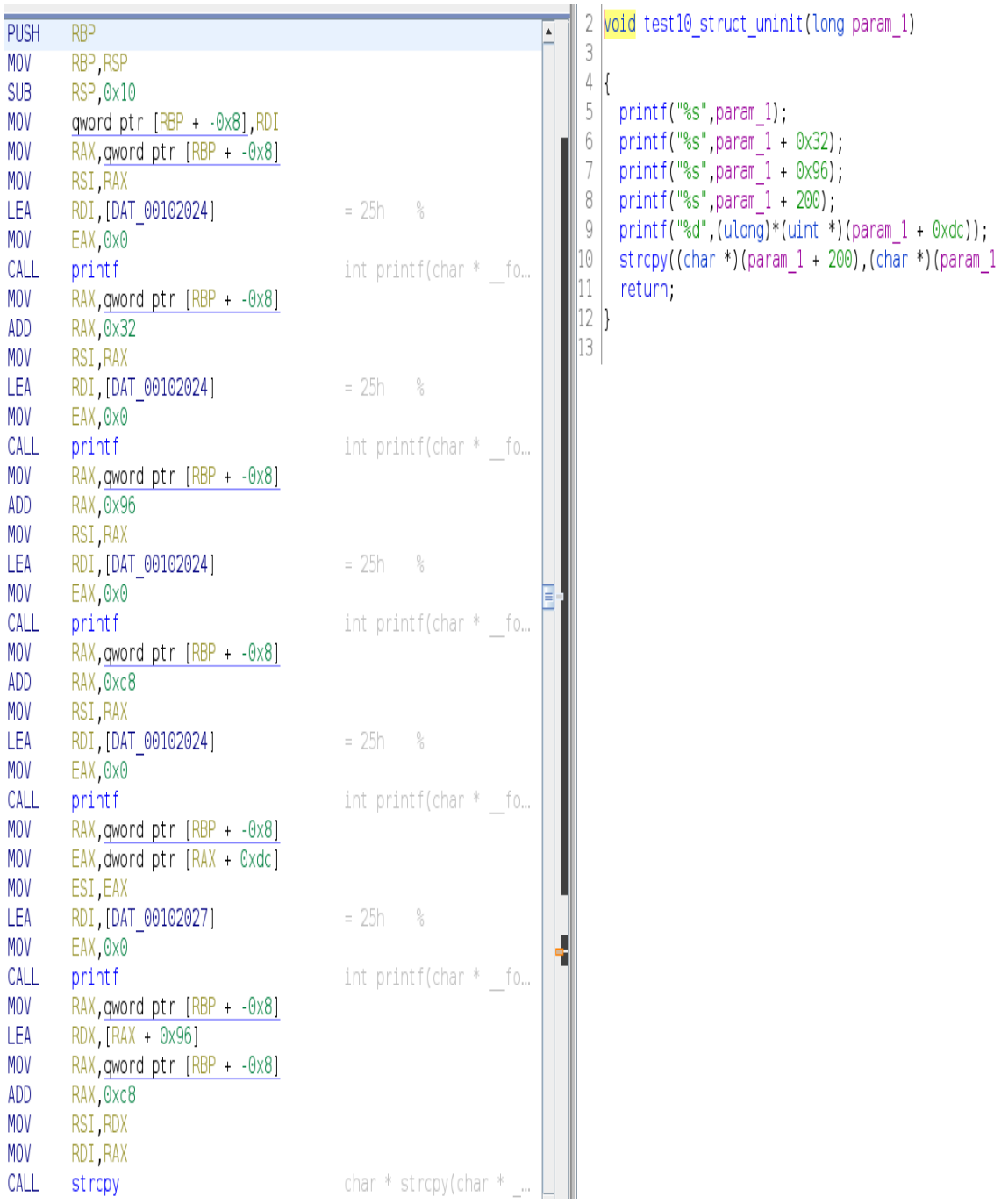


```
PUSH    RBP
MOV     RBP,RSP
SUB     RSP,0x10
MOV     qword ptr [RBP + -0x8],RDI
MOV     RAX,qword ptr [RBP + -0x8]
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0x32
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0x96
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0xc8
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
MOV     EAX,dword ptr [RAX + 0xdc]
MOV     ESI,EAX
LEA     RDI,[DAT_00102027]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
LEA     RDX,[RAX + 0x32]
MOV     RAX,qword ptr [RBP + -0x8]
MOV     RSI,RDX
MOV     RDI,RAX
CALL    strcpy                          char * strcpy(char * _…
NOP
LEAVE
```

```
2  void test9_struct_init(char *param_1)
3
4  {
5    printf("%s",param_1);
6    printf("%s",param_1 + 0x32);
7    printf("%s",param_1 + 0x96);
8    printf("%s",param_1 + 200);
9    printf("%d",(ulong)*(uint *)(param_1 + 0xdc));
10   strcpy(param_1,param_1 + 0x32);
11   return;
12 }
13
```

Figure E.9. Test 9: Function with initialized struct pointer

Test 9 passed an initialized stuct pointer to a function that used *printf()* and *strcpy()*. Figure E.9 showed how internal struct data types are passed as parameters. At the beginning of the disassembly, it is seen that the pointer to the struct is stored in a local variable. Just before the *strcpy()* call, the struct pointer is then calculated for each data type that is used as a pointer.

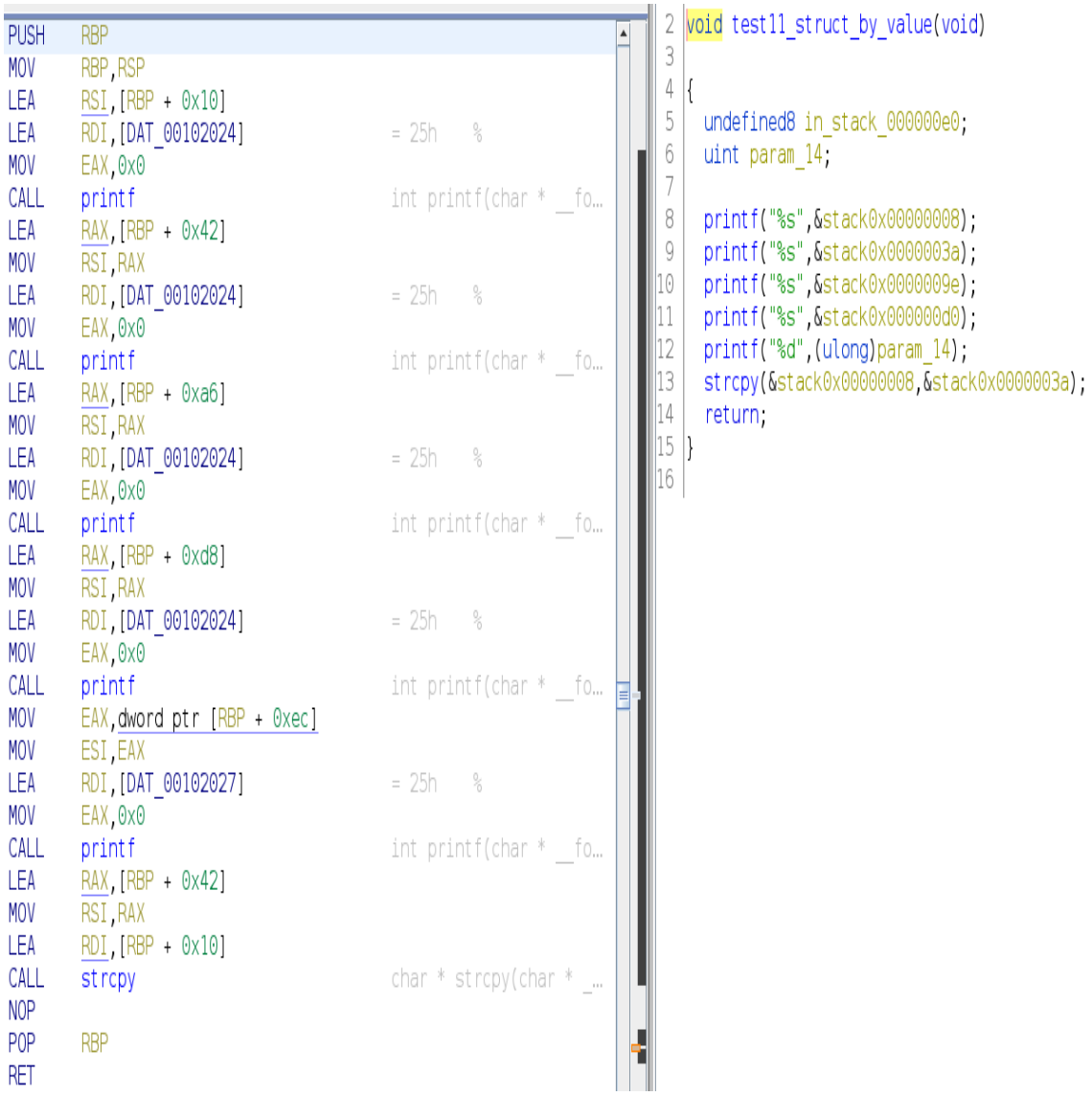


```
PUSH    RBP
MOV     RBP,RSP
SUB     RSP,0x10
MOV     qword ptr [RBP + -0x8],RDI
MOV     RAX,qword ptr [RBP + -0x8]
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0x32
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0x96
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0xc8
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
MOV     EAX,dword ptr [RAX + 0xdc]
MOV     ESI,EAX
LEA     RDI,[DAT_00102027]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo…
MOV     RAX,qword ptr [RBP + -0x8]
LEA     RDX,[RAX + 0x96]
MOV     RAX,qword ptr [RBP + -0x8]
ADD     RAX,0xc8
MOV     RSI,RDX
MOV     RDI,RAX
CALL    strcpy                          char * strcpy(char * _…
```

```
2  void test10_struct_uninit(long param_1)
3
4  {
5    printf("%s",param_1);
6    printf("%s",param_1 + 0x32);
7    printf("%s",param_1 + 0x96);
8    printf("%s",param_1 + 200);
9    printf("%d",(ulong)*(uint *)(param_1 + 0xdc));
10   strcpy((char *)(param_1 + 200),(char *)(param_1
11   return;
12 }
13
```

Figure E.10. Test 10: Function with uninitialized struct pointer

Test 10 passes an uninitialized struct pointer to a function that uses *printf()* and *strcpy()*. Figure E.10 shows that uninitialized structs are handled the same way as initialized structs.



```
PUSH    RBP
MOV     RBP,RSP
LEA     RSI,[RBP + 0x10]
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
LEA     RAX,[RBP + 0x42]
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
LEA     RAX,[RBP + 0xa6]
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
LEA     RAX,[RBP + 0xd8]
MOV     RSI,RAX
LEA     RDI,[DAT_00102024]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
MOV     EAX,dword ptr [RBP + 0xec]
MOV     ESI,EAX
LEA     RDI,[DAT_00102027]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
LEA     RAX,[RBP + 0x42]
MOV     RSI,RAX
LEA     RDI,[RBP + 0x10]
CALL    strcpy                          char * strcpy(char * _...
NOP
POP     RBP
RET
```

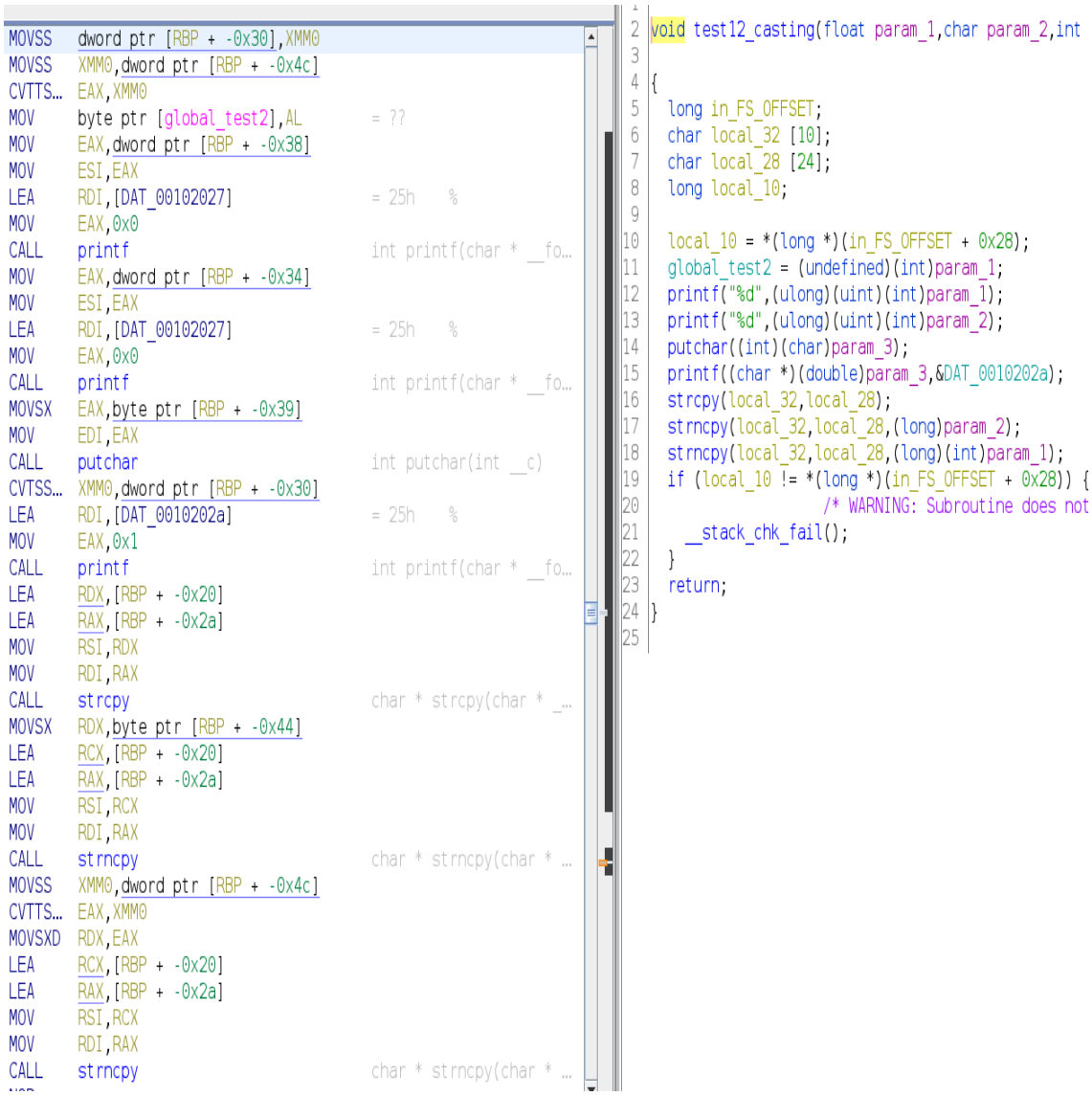
```
2  void test11_struct_by_value(void)
3
4  {
5    undefined8 in_stack_000000e0;
6    uint param_14;
7
8    printf("%s",&stack0x00000008);
9    printf("%s",&stack0x0000003a);
10   printf("%s",&stack0x0000009e);
11   printf("%s",&stack0x000000d0);
12   printf("%d",(ulong)param_14);
13   strcpy(&stack0x00000008,&stack0x0000003a);
14   return;
15 }
16
```

Figure E.11. Test 11: Function with struct passed by value

Test 11 passed a struct by values on the stack to a function that used *printf()* and *strcpy()*. Figure E.11 shows the disassembly and how structs passed by values on the stack are used in functions. Looking just before the *strcpy()* call, it is noted that the parameters are directly referenced in the stack. This is seen by the positive offsets to RBP for the parameter passed to *strcpy()*. Figure E.12 shows how parameters are put on the stack. In this case the values are pushed onto the stack; however, depending on the stack layout these values may be moved instead of pushed onto the stack.

```
PUSH     qword ptr [RBP + -0x98]
PUSH     qword ptr [RBP + -0xa0]
PUSH     qword ptr [RBP + -0xa8]
PUSH     qword ptr [RBP + -0xb0]
PUSH     qword ptr [RBP + -0xb8]
PUSH     qword ptr [RBP + -0xc0]
PUSH     qword ptr [RBP + -0xc8]
PUSH     qword ptr [RBP + -0xd0]
PUSH     qword ptr [RBP + -0xd8]
PUSH     qword ptr [RBP + -0xe0]
PUSH     qword ptr [RBP + -0xe8]
PUSH     qword ptr [RBP + -0xf0]
PUSH     qword ptr [RBP + -0xf8]
PUSH     qword ptr [RBP + -0x100]
PUSH     qword ptr [RBP + -0x108]
PUSH     qword ptr [RBP + -0x110]
PUSH     qword ptr [RBP + -0x118]
PUSH     qword ptr [RBP + -0x120]
PUSH     qword ptr [RBP + -0x128]
PUSH     qword ptr [RBP + -0x130]
PUSH     qword ptr [RBP + -0x138]
PUSH     qword ptr [RBP + -0x140]
PUSH     qword ptr [RBP + -0x148]
PUSH     qword ptr [RBP + -0x150]
PUSH     qword ptr [RBP + -0x158]
PUSH     qword ptr [RBP + -0x160]
PUSH     qword ptr [RBP + -0x168]
PUSH     qword ptr [RBP + -0x170]
CALL     test11_struct_by_value
```

Figure E.12. Test 11: Function that calls with a struct passed by value



```
MOVSS   dword ptr [RBP + -0x30],XMM0
MOVSS   XMM0,dword ptr [RBP + -0x4c]
CVTTS...EAX,XMM0
MOV     byte ptr [global_test2],AL      = ??
MOV     EAX,dword ptr [RBP + -0x38]
MOV     ESI,EAX
LEA     RDI,[DAT_00102027]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
MOV     EAX,dword ptr [RBP + -0x34]
MOV     ESI,EAX
LEA     RDI,[DAT_00102027]              = 25h    %
MOV     EAX,0x0
CALL    printf                          int printf(char * __fo...
MOVSX   EAX,byte ptr [RBP + -0x39]
MOV     EDI,EAX
CALL    putchar                         int putchar(int __c)
CVTSS...XMM0,dword ptr [RBP + -0x30]
LEA     RDI,[DAT_0010202a]              = 25h    %
MOV     EAX,0x1
CALL    printf                          int printf(char * __fo...
LEA     RDX,[RBP + -0x20]
LEA     RAX,[RBP + -0x2a]
MOV     RSI,RDX
MOV     RDI,RAX
CALL    strcpy                          char * strcpy(char * _...
MOVSX   RDX,byte ptr [RBP + -0x44]
LEA     RCX,[RBP + -0x20]
LEA     RAX,[RBP + -0x2a]
MOV     RSI,RCX
MOV     RDI,RAX
CALL    strncpy                         char * strncpy(char * ...
MOVSS   XMM0,dword ptr [RBP + -0x4c]
CVTTS...EAX,XMM0
MOVSXD  RDX,EAX
LEA     RCX,[RBP + -0x20]
LEA     RAX,[RBP + -0x2a]
MOV     RSI,RCX
MOV     RDI,RAX
CALL    strncpy                         char * strncpy(char * ...
```

```
2  void test12_casting(float param_1,char param_2,int
3
4  {
5    long in_FS_OFFSET;
6    char local_32 [10];
7    char local_28 [24];
8    long local_10;
9
10   local_10 = *(long *)(in_FS_OFFSET + 0x28);
11   global_test2 = (undefined)(int)param_1;
12   printf("%d",(ulong)(uint)(int)param_1);
13   printf("%d",(ulong)(uint)(int)param_2);
14   putchar((int)(char)param_3);
15   printf((char *)(double)param_3,&DAT_0010202a);
16   strcpy(local_32,local_28);
17   strncpy(local_32,local_28,(long)param_2);
18   strncpy(local_32,local_28,(long)(int)param_1);
19   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
20                     /* WARNING: Subroutine does not
21     __stack_chk_fail();
22   }
23   return;
24 }
25
```

Figure E.13. Test 12: Function with casting

Test 12 passed a char, an int, and a float to a function that cast the parameters to local variables and used *printf()*, *strcpy()*, and *strncpy()*. Figure E.13 shows the various ways that casting is handled. These cases show that casting is performed by individual instructions, which store the result as the proper data type.

APPENDIX F:
Script Outputs

This appendix describes the outputs produced by the buffer overflow script. The first part covers the outputs to the console. The second part covers the outputs to the CSV files.

F.1 Console Outputs

The console outputs consist of the Ghidra's version information, messages, and source information that causes a particular sink to overflow. A message is either a warning message or a caution message. Warning messages also include the number of bytes that the buffer is overflowed. Following the message is information about the sources that caused the overflow. Information about a source includes its sink, argument at the sink, source name, source's function name, sink's location, source's location, the source's path, size, max fill, usage, and extra equations. There can be many messages per each run of the script. The information listed below is a sample output from the script.

```
This script is intended to be run on Ghidra version 9.1-BETA
You are currently running Ghidra 9.1-BETA


Warning! The source max fill size is larger than the allocated
size for the destination.
There is an over flow of: 49
Source Info
    Sink: strncat
    Argument: charptr1
    Name: [RBP + -0xb0]
    Funcion: CWE121_Stack_Based_Buffer_Overflow
    __CWE806_char_declare_ncat_42_bad
    Ref Location: 0010182a
    Var Location: 0010181d
    Path: [[CWE121_Stack_Based_Buffer_Overflow
```

```
    __CWE806_char_declare_ncat_42_bad, 0010182a]]
    Size: 50
    Max Fill: 99
    Usage: ['0010182a CALL 0x00101090 using RDI',
    '0010182a CALL 0x00101090 using RDI']
    Extra: None
Source Info
    Sink: strncat
    Argument: charptr2
    Name: [RBP + -0x70]
    Funcion: CWE121_Stack_Based_Buffer_
    Overflow__CWE806_char_declare_ncat_42_bad
    Ref Location: 0010182a
    Var Location: 0010179b
    Path: [[CWE121_Stack_Based_Buffer_Overflow
    __CWE806_char_declare_ncat_42_bad, 0010182a]]
    Size: 104
    Max Fill: 99
    Usage: ['001017b0 CALL 0x0010174e using RDI',
    '0010180e CALL 0x00101050 using RDI',
    '0010182a CALL 0x00101090 using RSI',
    '0010183d CALL 0x00101205 using RDI',
    '00101852 CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '0010176b CALL 0x00101080 using RDI',
    '001017b0 CALL 0x0010174e using RDI',
    '0010180e CALL 0x00101050 using RDI',
    '0010182a CALL 0x00101090 using RSI',
    '0010183d CALL 0x00101205 using RDI',
    '00101852 CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '0010176b CALL 0x00101080 using RDI']
    Extra: None
Source Info
    Sink: strncat
```

```
    Argument: int
    Name: [RBP + -0x70]
    Funcion: CWE121_Stack_Based_Buffer_Overflow
    __CWE806_char_declare_ncat_42_bad
    Ref Location: 0010182a
    Var Location: 0010180e
    Path: [[CWE121_Stack_Based_Buffer_Overflow
    __CWE806_char_declare_ncat_42_bad, 0010182a]]
    Size: 104
    Max Fill: 99
    Usage: ['001017b0 CALL 0x0010174e using RDI',
    '0010180e CALL 0x00101050 using RDI',
    '0010182a CALL 0x00101090 using RSI',
    '0010183d CALL 0x00101205 using RDI',
    '00101852 CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '0010176b CALL 0x00101080 using RDI',
    '001017b0 CALL 0x0010174e using RDI',
    '0010180e CALL 0x00101050 using RDI',
    '0010182a CALL 0x00101090 using RSI',
    '0010183d CALL 0x00101205 using RDI',
    '00101852 CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '0010176b CALL 0x00101080 using RDI']
    Extra:

Caution! The number of bytes to write is larger than the
allocated size for the destination.
Source Info
    Sink: strncat
    Argument: charptr1
    Name: [RBP + -0xb0]
    Funcion: goodG2B
    Ref Location: 00101935
    Var Location: 00101928
```

```
    Path: [[goodG2B, 00101935]]
    Size: 50
    Max Fill: 49
    Usage: ['00101935 CALL 0x00101090 using RDI',
    '00101935 CALL 0x00101090 using RDI']
    Extra: None
Source Info
    Sink: strncat
    Argument: charptr2
    Name: [RBP + -0x70]
    Funcion: goodG2B
    Ref Location: 00101935
    Var Location: 001018a6
    Path: [[goodG2B, 00101935]]
    Size: 104
    Max Fill: 49
    Usage: ['001018bb CALL 0x00101859 using RDI',
    '00101919 CALL 0x00101050 using RDI',
    '00101935 CALL 0x00101090 using RSI',
    '00101948 CALL 0x00101205 using RDI',
    '0010195d CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '00101876 CALL 0x00101080 using RDI',
    '001018bb CALL 0x00101859 using RDI',
    '00101919 CALL 0x00101050 using RDI',
    '00101935 CALL 0x00101090 using RSI',
    '00101948 CALL 0x00101205 using RDI',
    '0010195d CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '00101876 CALL 0x00101080 using RDI']
    Extra: None
Source Info
    Sink: strncat
    Argument: int
    Name: [RBP + -0x70]
```

```
    Funcion: goodG2B
    Ref Location: 00101935
    Var Location: 00101919
    Path: [[goodG2B, 00101935]]
    Size: 104
    Max Fill: 49
    Usage: ['001018bb CALL 0x00101859 using RDI',
    '00101919 CALL 0x00101050 using RDI',
    '00101935 CALL 0x00101090 using RSI',
    '00101948 CALL 0x00101205 using RDI',
    '0010195d CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '00101876 CALL 0x00101080 using RDI',
    '001018bb CALL 0x00101859 using RDI',
    '00101919 CALL 0x00101050 using RDI',
    '00101935 CALL 0x00101090 using RSI',
    '00101948 CALL 0x00101205 using RDI',
    '0010195d CALL 0x00101060 using qword ptr [RBP + -0xb8]',
    '0010122b CALL 0x00101070 using RSI',
    '00101876 CALL 0x00101080 using RDI']
    Extra:
ghidra_overflow_detection.py> Finished!
```

F.2 CSV Outputs

The ODSS script produces two output CSV files. The first file contains all the sources in the binary file. The second file contains all the sources that can cause an overflow in the binary file. The columns in both CSV files are a message type, sink name, argument at the sink, source name, source's function, sink's location, source's location, path, size, and max fill. Figure F.1 is an example CSV output file of the sources that can cause an overflow.

	A	B	C	D	E	F	G	H	I	J
1	Type	Sink	Argument	Src Name	Function Found	Sink Location	Argument Location	Path	Size	Max Fill
2	Warning	strncat	charptr1	[RBP + -0xb0]	CWE121_Stack_Base	0x0010182a	0x0010181d	[[CWE121_Stack_	50	99
3	Warning	strncat	charptr2	[RBP + -0x70]	CWE121_Stack_Base	0x0010182a	0x0010179b	[[CWE121_Stack_	104	99
4	Warning	strncat	int	[RBP + -0x70]	CWE121_Stack_Base	0x0010182a	0x0010180e	[[CWE121_Stack_	104	99
5	Caution	strncat	charptr1	[RBP + -0xb0]	goodG2B	0x00101935	0x00101928	[[goodG2B, 0010	50	49
6	Caution	strncat	charptr2	[RBP + -0x70]	goodG2B	0x00101935	0x001018a6	[[goodG2B, 0010	104	49
7	Caution	strncat	int	[RBP + -0x70]	goodG2B	0x00101935	0x00101919	[[goodG2B, 0010	104	49

Figure F.1. CSV file containing the sources that can overflow a source

APPENDIX G:
Functional Test Cases

This appendix summarizes the test cases from the Juliet test suite that are used for this thesis. Each test case is described in terms of a functional variant, a sink, a flow variant, a character type, and an expected result. The functional variant[3] is the type of vulnerability in the code. The sink is the *libc* function that can cause an overflow. The flow variant[4] is the type of control flow that occurs in the program. The character type is the size of the characters used in the sources. A char is a 1-byte character and a wide is a 4-byte character. The expected result is the type of message that the script should produce based on the assembly code. The test cases are named based on the program name found in the Juliet test suite. The Juliet test suite follows a standard naming convention for its files. The first part is the CWE ID followed by the shortened CWE entry name. This is proceeded by the functional variant name and the flow variant. The functional variant name is considered the flaw type and the flow variant is a number to represent the control flow used in the program (refer to Appendix B for the flow variant list). The last identifier is the programming language.

Example:

CWE Entry ID: 121

Shortened CWE Entry Name: Stack Based Buffer Overflow

Functional Variant: CWE805_char_declare_memcpy

Flow Variant: 07

Language: C

This creates the file:

CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_07.c

[3]Naming of the functional variants is based on the Juliet test suite naming convention [3].

[4]Naming of the flow variants is based on the Juliet test suite naming convention [3].

For each test case there is a table summarizing the sinks that are tested. The rows of the table are the sinks found in the program. The columns are detailed below:

- Code: Programming language for the sink. Either C or Assembly.
- Juliet Function: The name of function where the sink is found. The Juliet test suite names a function 'bad' if the function can cause a buffer overflow and names a function 'good' if the function cannot produce a buffer overflow.
- Dst: The sink's destination parameter.
- Src size: The allocated size of the sink's source parameter.
- Src fill: The largest string length for the sink's source parameter.
- Amount: The sink's integer parameter or the number of bytes to copy. (This column is not present for all sinks)
- Result: The type of message that should be produced and the number of bytes that should overflow.

To recapitulate, if the largest string length for the source parameter is larger than the allocated size of the destination parameter, then a warning message is produced. For the sinks that require three arguments, the maximum fill of the integer parameter indicating the amount of bytes to copy must be larger than the allocated size of the destination parameter. If the allocated size of the source parameter is larger than allocated size of the destination parameter, then a caution message is produced. The three-argument case also requires the amount parameter to be larger than the allocated size of the destination parameter. Expected results are based on the assembly code. The script does not look at the C code, thus it does not produce messages for the C code. The C code information is displayed to show that there is a different result when comparing the C code to the assembly code.

Test 1: CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01

- Functional variant: Struct overrun - Incorrect length value used for a struct.
- Sink: *memcpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.1. Summary of Test Case 1

Code	Juliet Function	Dst	Src size	Src fill	Amount	Result
C	bad	16	32	32	32	Warning 16 bytes
C	good1	16	32	32	16	No Overflow
Assembly	bad	16	32	32	32	Warning 16 bytes
Assembly	good1	16	32	32	16	No Overflow

Test 2: CWE121_Stack_Based_Buffer_Overflow__wchar_t_type_overrun_memmove_08

- Functional variant: Struct overrun - Incorrect length value used for a struct.
- Sink: *memmove()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.

Table G.2. Summary of Test Case 2

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	64	128	128	80	Warning 16 bytes
C	good1	64	128	128	64	No Overflow
C	good2	64	128	128	64	No Overflow
Assembly	bad	64	128	128	80	Warning 16 bytes
Assembly	good1	64	128	128	64	No Overflow
Assembly	good2	64	128	128	64	No Overflow

Test 3: CWE121_Stack_Based_Buffer_Overflow__CWE193_char_declare_cpy_02

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strcpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.

Table G.3. Summary of Test Case 3

Code	Function	Dst	Src size	Src fill	Result
C	bad	10	11	11	Warning 1 byte
C	good1	11	11	11	No Overflow
C	good2	11	11	11	No Overflow
Assembly	bad	10	11	11	Warning 1 byte
Assembly	good1	11	11	11	No Overflow
Assembly	good2	11	11	11	No Overflow

Test 4: CWE121_Stack_Based_Buffer_Overflow__CWE193_wchar_t_declare_cpy_31

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcscpy()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one caution message. Src size is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.4. Summary of Test Case 4

Code	Function	Dst	Src size	Src fill	Result
C	bad	40	44	44	Warning 4 bytes
C	good1	44	44	44	No Overflow
Assembly	bad	48	56	44	No Overflow
Assembly	good1	48	56	44	No Overflow

Test 5: CWE121_Stack_Based_Buffer_Overflow__CWE193_char_declare_cpy_34

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strcpy()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.5. Summary of Test Case 5

Code	Function	Dst	Src size	Src fill	Result
C	bad	10	11	11	Warning 1 byte
C	good1	11	11	11	No Overflow
Assembly	bad	10	11	11	Warning 1 byte
Assembly	good1	11	11	11	No Overflow

Test 6: CWE121_Stack_Based_Buffer_Overflow__CWE193_wchar_t_declare_cpy_41

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcscpy()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.6. Summary of Test Case 6

Code	Function	Dst	Src size	Src fill	Result
C	bad	40	44	44	Warning 4 bytes
C	good1	44	44	44	No Overflow
Assembly	bad	56	44	44	No Overflow
Assembly	good1	56	44	44	No Overflow

Test 7: CWE121_Stack_Based_Buffer_Overflow__CWE193_char_declare_cpy_51

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strcpy()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.7. Summary of Test Case 7

Code	Function	Dst	Src size	Src fill	Result
C	bad	10	11	11	Warning 1 byte
C	good1	11	11	11	No Overflow
Assembly	bad	11	11	11	No Overflow
Assembly	good1	11	11	11	No Overflow

Test 8: CWE121_Stack_Based_Buffer_Overflow__CWE193_wchar_t_declare_cpy_63

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcscpy()*
- Flow variant: Pointer to data passed from one function to another in different source files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.8. Summary of Test Case 8

Code	Function	Dst	Src size	Src fill	Result
C	bad	40	44	44	Warning 4 bytes
C	good1	44	44	44	No Overflow
Assembly	bad	56	44	44	No Overflow
Assembly	good1	56	44	44	No Overflow

Test 9: CWE121_Stack_Based_Buffer_Overflow__CWE193_char_declare_memcpy_03

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memcpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.

Table G.9. Summary of Test Case 9

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	10	11	11	11	Warning 1 byte
C	good1	11	11	11	11	No Overflow
C	good2	11	11	11	11	No Overflow
Assembly	bad	10	11	11	11	Warning 1 byte
Assembly	good1	11	11	11	11	No Overflow
Assembly	good2	11	11	11	11	No Overflow

Test 10: CWE121_Stack_Based_Buffer_Overflow__CWE193_char_declare_memmove_32

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memmove()*
- Flow variant: Uses pointers to point to the source values.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.10. Summary of Test Case 10

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	10	11	11	11	Warning 1 byte
C	good1	11	11	11	11	No Overflow
Assembly	bad	10	11	11	11	Warning 1 byte
Assembly	good1	11	11	11	11	No Overflow

Test 11: CWE121_Stack_Based_Buffer_Overflow__CWE193_char_declare_ncpy_34

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strncpy()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.11. Summary of Test Case 11

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	10	11	11	11	Warning 1 byte
C	good1	11	11	11	11	No Overflow
Assembly	bad	10	11	11	11	Warning 1 byte
Assembly	good1	11	11	11	11	No Overflow

Test 12: CWE121_Stack_Based_Buffer_Overflow__CWE193_wchar_t_declare_ncpy_44

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcsncpy()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.12. Summary of Test Case 12

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	40	44	44	44	Warning 4 bytes
C	good1	44	44	44	44	No Overflow
Assembly	bad	56	44	44	44	No Overflow
Assembly	good1	56	44	44	44	No Overflow

Test 13: CWE121_Stack_Based_Buffer_Overflow__CWE193_wchar_t_declare_memcpy_52

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memcpy()*
- Flow variant: Passes source values between files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.13. Summary of Test Case 13

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	40	44	44	44	Warning 4 bytes
C	good1	44	44	44	44	No Overflow
Assembly	bad	56	44	44	44	No Overflow
Assembly	good1	56	44	44	44	No Overflow

Test 14: CWE121_Stack_Based_Buffer_Overflow__CWE193_wchar_t_declare_memmove_64

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memmove()*
- Flow variant: Void pointer to data passed from one function to another in different source files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.14. Summary of Test Case 14

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	40	44	44	44	Warning 4 bytes
C	good1	44	44	44	44	No Overflow
Assembly	bad	56	44	44	44	No Overflow
Assembly	good1	56	44	44	44	No Overflow

Test 15: CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_ncat_10

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *strncat()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.

Table G.15. Summary of Test Case 15

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	100	100	100	100	No Overflow
C	good2	100	100	100	100	No Overflow
Assembly	bad	64	100	100	100	Warning 36 bytes
Assembly	good1	112	100	100	100	No Overflow
Assembly	good2	112	100	100	100	No Overflow

Test 16: CWE121_Stack_Based_Buffer_Overflow__CWE805_wchar_t_declare_ncat_31

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *wcsncat()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.16. Summary of Test Case 16

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	200	400	400	400	Warning 200 bytes
C	good1	400	400	400	400	No Overflow
Assembly	bad	208	400	400	400	Warning 192 bytes
Assembly	good1	400	400	400	400	No Overflow

Test 17: CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memmove_34

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *memmove()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.17. Summary of Test Case 17

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	100	100	100	100	No Overflow
Assembly	bad	64	100	100	100	Warning 36 bytes
Assembly	good1	100	100	100	100	No Overflow

Test 18: CWE121_Stack_Based_Buffer_Overflow__CWE805_wchar_t_declare_memcpy_44

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *memcpy()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.18. Summary of Test Case 18

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	200	400	400	400	Warning 200 bytes
C	good1	400	400	400	400	No Overflow
Assembly	bad	408	400	400	400	No Overflow
Assembly	good1	408	400	400	400	No Overflow

Test 19: CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_ncat_53

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *strncpy()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.19. Summary of Test Case 19

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	100	100	100	100	No Overflow
Assembly	bad	104	100	100	100	No Overflow
Assembly	good1	104	100	100	100	No Overflow

Test 20: CWE121_Stack_Based_Buffer_Overflow__CWE805_wchar_t_declare_ncpy_65

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *wcsncpy()*
- Flow variant: Data passed as an argument from one function to a function in a different source file called via a function pointer
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.20. Summary of Test Case 20

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	200	400	400	400	Warning 200 bytes
C	good1	400	400	400	400	No Overflow
Assembly	bad	408	400	400	400	No Overflow
Assembly	good1	408	400	400	400	No Overflow

Test 21: CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_declare_ncpy_11

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *wcsncpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
  - Assembly good2 function: Produces one caution message. Src size is greater than Dst.

Table G.21. Summary of Test Case 21

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	200	400	400	400	Warning 200 bytes
C	good1	200	400	200	200	Caution 200 bytes
C	good2	200	400	200	200	Caution 200 bytes
Assembly	bad	200	408	400	400	Warning 200 bytes
Assembly	good1	200	408	200	200	Caution 208 bytes
Assembly	good2	200	408	200	200	Caution 208 bytes

Test 22: CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_ncpy_22

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *strncpy()*
- Flow variant: Uses conditional to control the flow of the program and passes source information between functions.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
  - Assembly good2 function: Produces one caution message. Src size is greater than Dst.

Table G.22. Summary of Test Case 22

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	50	100	50	50	Caution 50 bytes
C	good2	50	100	50	50	Caution 50 bytes
Assembly	bad	50	104	100	100	Warning 50 bytes
Assembly	good1	50	104	50	50	Caution 54 bytes
Assembly	good2	50	104	50	50	Caution 54 bytes

Test 23: CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_declare_memcpy_32

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *memcpy()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.23. Summary of Test Case 23

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	200	400	400	400	Warning 200 bytes
C	good1	200	400	200	200	Caution 200 bytes
Assembly	bad	200	408	400	400	Warning 200 bytes
Assembly	good1	200	408	200	200	Caution 208 bytes

Test 24: CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_declare_ncat_34

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *wcsncat()*
- Flow variant: Uses a union for the source parameter.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.24. Summary of Test Case 24

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	200	400	400	400	Warning 200 bytes
C	good1	200	400	200	200	Caution 200 bytes
Assembly	bad	200	408	400	400	Warning 200 bytes
Assembly	good1	200	408	200	200	Caution 208 bytes

Test 25: CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_ncat_42

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *strncpy()*
- Flow variant: Passes source information between functions.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.25. Summary of Test Case 25

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	50	100	50	50	Caution 50 bytes
Assembly	bad	50	104	100	100	Warning 50 bytes
Assembly	good1	50	104	50	50	Caution 54 bytes

Test 26: CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_memmove_54

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *memmove()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.26. Summary of Test Case 26

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	50	100	50	50	Caution 50 bytes
Assembly	bad	50	104	100	100	Warning 50 bytes
Assembly	good1	50	104	50	50	Caution 54 bytes

Test 27: CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_memcpy_66

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *memcpy()*
- Flow variant: Data passed in an array from one function to another in different source files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.27. Summary of Test Case 27

Code	Function	Dst	Src size	Src fill	Amount	Result
C	bad	50	100	100	100	Warning 50 bytes
C	good1	50	100	50	50	Caution 50 bytes
Assembly	bad	50	104	100	100	Warning 50 bytes
Assembly	good1	50	104	50	50	Caution 54 bytes

Test 28: CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cat_17

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.28. Summary of Test Case 28

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	100	100	100	No Overflow
Assembly	bad	64	100	100	Warning 36 bytes
Assembly	good1	112	100	100	No Overflow

Test 29: CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_declare_cat_31

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *wcscat()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.29. Summary of Test Case 29

Code	Function	Dst	Src size	Src fill	Result
C	bad	200	400	400	Warning 200 bytes
C	good1	400	400	400	No Overflow
Assembly	bad	208	400	400	Warning 192 bytes
Assembly	good1	400	400	400	No Overflow

Test 30: CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_declare_cpy_34

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *wcscpy()*
- Flow variant: Uses a union for the source parameter.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.

Table G.30. Summary of Test Case 30

Code	Function	Dst	Src size	Src fill	Result
C	bad	200	400	400	Warning 200 bytes
C	good1	400	400	400	No Overflow
Assembly	bad	208	400	400	Warning 192 bytes
Assembly	good1	400	400	400	No Overflow

Test 31: CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cpy_41

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *strcpy()*
- Flow variant: Passes source information between functions.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.31. Summary of Test Case 31

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	100	100	100	No Overflow
Assembly	bad	104	100	100	No Overflow
Assembly	good1	104	100	100	No Overflow

Test 32: CWE121_Stack_Based_Buffer_Overflow__dest_wchar_t_declare_cat_51

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *wcscpy()*
- Flow variant: Passes source values between files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.32. Summary of Test Case 32

Code	Function	Dst	Src size	Src fill	Result
C	bad	200	400	400	Warning 200 bytes
C	good1	400	400	400	No Overflow
Assembly	bad	408	400	400	No Overflow
Assembly	good1	408	400	400	No Overflow

Test 33: CWE121_Stack_Based_Buffer_Overflow__dest_char_declare_cat_67

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Data passed in a struct from one function to another in different source files.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.

Table G.33. Summary of Test Case 33

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	100	100	100	No Overflow
Assembly	bad	104	100	100	No Overflow
Assembly	good1	104	100	100	No Overflow

Test 34: CWE121_Stack_Based_Buffer_Overflow__src_wchar_t_declare_cpy_18

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *wcscpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.34. Summary of Test Case 34

Code	Function	Dst	Src size	Src fill	Result
C	bad	200	400	400	Warning 200 bytes
C	good1	200	400	200	Caution 200 bytes
Assembly	bad	208	408	400	Warning 192 bytes
Assembly	good1	208	408	200	Caution 200 bytes

Test 35: CWE121_Stack_Based_Buffer_Overflow__src_wchar_t_declare_cat_21

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *wcscat()*
- Flow variant: Uses conditional to control the flow of the program and passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
  - Assembly good2 function: Produces one caution message. Src size is greater than Dst.

Table G.35. Summary of Test Case 35

Code	Function	Dst	Src size	Src fill	Result
C	bad	200	400	400	Warning 200 bytes
C	good1	200	400	200	Caution 200 bytes
C	good2	200	400	200	Caution 200 bytes
Assembly	bad	208	408	400	Warning 192 bytes
Assembly	good1	208	408	200	Caution 200 bytes
Assembly	good2	208	408	200	Caution 200 bytes

Test 36: CWE121_Stack_Based_Buffer_Overflow__src_char_declare_cpy_32

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcpy()*
- Flow variant: Uses pointers to point to the source values.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.36. Summary of Test Case 36

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	50	100	50	Caution 50 bytes
Assembly	bad	64	104	100	Warning 36 bytes
Assembly	good1	64	104	50	Caution 40 bytes

Test 37: CWE121_Stack_Based_Buffer_Overflow__src_char_declare_cat_34

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.37. Summary of Test Case 37

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	50	100	50	Caution 50 bytes
Assembly	bad	64	104	100	Warning 36 bytes
Assembly	good1	64	104	50	Caution 40 bytes

Test 38: CWE121_Stack_Based_Buffer_Overflow__src_wchar_t_declare_cat_42

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *wcscat()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.38. Summary of Test Case 38

Code	Function	Dst	Src size	Src fill	Result
C	bad	200	400	400	Warning 200 bytes
C	good1	200	400	200	Caution 200 bytes
Assembly	bad	208	408	400	Warning 192 bytes
Assembly	good1	208	408	200	Caution 200 bytes

Test 39: CWE121_Stack_Based_Buffer_Overflow__src_char_declare_cat_54

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.39. Summary of Test Case 39

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	50	100	50	Caution 50 bytes
Assembly	bad	50	104	100	Warning 50 bytes
Assembly	good1	50	104	50	Caution 54 bytes

Test 40: CWE121_Stack_Based_Buffer_Overflow__src_char_declare_cpy_68

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcpy()*
- Flow variant: Data passed as a global variable from one function to another in different source files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.

Table G.40. Summary of Test Case 40

Code	Function	Dst	Src size	Src fill	Result
C	bad	50	100	100	Warning 50 bytes
C	good1	50	100	50	Caution 50 bytes
Assembly	bad	50	104	100	Warning 50 bytes
Assembly	good1	50	104	50	Caution 54 bytes

APPENDIX H:
Test Case Results

This appendix contains a summary of the outputs generated by the ODSS script. The test cases show either a warning or caution message with values that were found for the given parameters. Each test case is described in terms of a functional variant, a sink, a flow variant, a character type, an expected result, and an actual result. The functional variant[5] is the type of vulnerability in the code. The sink is the *libc* function that can cause an overflow. The flow variant[6] is the type of control flow that occurs in the program. The character type is the size of the characters used in the sources. A char is a 1-byte character and a wide is a 4-byte character. The expected result is the type of message that the script should produce based on the assembly code. The actual result is the message(s) and sources that caused the overflow. When a test has a buffer overflow, the results are summarized in a table. The rows are the sources for a given sink. The columns are described below:

- Msg: The message number with which the source is associated.
- Sink: The name of the sink that caused the overflow.
- Sink Location: The sink's address in the assembly code.
- Argument: The source's argument position in the sink.
- Size: The source's allocated size.
- Max Fill: The source's largest string length.

To recapitulate, if the largest string length for the source parameter is larger than the allocated size of the destination parameter, then a warning message is produced. For the sinks that require three arguments, the maximum fill of the integer parameter indicating the amount of bytes to copy must be larger than the allocated size of the destination parameter. If the allocated size of the source parameter is larger than allocated size of the destination parameter, then a caution message is produced. The three-argument case also requires the amount parameter to be larger than the allocated size of the destination parameter. Expected and actual results are based on the assembly code. The script does not look at the C code, thus it does not produce messages for the C code.

[5]Naming of the functional variants is based on the Juliet test suite naming convention [3].
[6]Naming of the flow variants is based on the Juliet test suite naming convention [3].

Test 1 Results:

- Functional variant: Struct overrun - Incorrect length value used for a struct.
- Sink: *memcpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.1. Summary of the results from test case 1

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memcpy	0x0010176f	charptr1	15	32
1	memcpy	0x0010176f	charptr2	31	31
1	memcpy	0x0010176f	int	32	32

Test 2 Results:

- Functional variant: Struct overrun - Incorrect length value used for a struct.
- Sink: *memmove()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one warning message (Msg 2). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good2 function: Produced one warning message (Msg 3). The max fill for arguments charptr2 and int are greater than the size of charptr1.

Table H.2. Summary of the results from test case 2

Warnings					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memmove	0x00101793	charptr1	60	80
1	memmove	0x00101793	charptr2	124	124
1	memmove	0x00101793	int	80	80
2	memmove	0x0010182b	charptr1	60	64
2	memmove	0x0010182b	charptr2	124	124
2	memmove	0x0010182b	int	64	64
3	memmove	0x001018b5	charptr1	60	64
3	memmove	0x001018b5	charptr2	124	124
3	memmove	0x001018b5	int	64	64

Test Case 2 produced a warning for two good implementations of the sink. This is caused by the destination parameter not accounting for the null byte at the end of the char array. This can be fixed by adding extra checks for the presence of a null bytes after a char array.

Test 3 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strcpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 is greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.
  - Assembly good2 function: Produced no messages.

Table H.3. Summary of the results from test case 3

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcpy	0x0010177a	charptr1	10	0
1	strcpy	0x0010177a	charptr2	11	11

Test 4 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcscpy()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one caution message. Src size is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one caution message (Msg 1). The size for charptr2 is greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.4. Summary of the results from test case 4

Cautions					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcscpy	0x001017c1	charptr1	48	44
1	wcscpy	0x001017c1	charptr2	56	44
2	wcscpy	0x0010187c	charptr1	48	44
2	wcscpy	0x0010187c	charptr2	56	44

Test 5 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strcpy()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 is greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.5. Summary of the results from test case 5

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcpy	0x0010178a	charptr1	10	11
1	strcpy	0x0010178a	charptr2	11	11

Test 6 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcscpy()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 7 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strcpy()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 8 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcscpy()*
- Flow variant: Pointer to data passed from one function to another in different source files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 9 Results: Produced one warning message. Expected one warning message.

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memcpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.
  - Assembly good2 function: Produced no messages.

Table H.6. Summary of the results from test case 9

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memcpy	0x0010179a	charptr1	10	0
1	memcpy	0x0010179a	charptr2	11	11
1	memcpy	0x0010179a	int	12	11

Test 10 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memmove()*
- Flow variant: Uses pointers to point to the source values.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one warning message (Msg 2). The max fill for arguments charptr2 and int are greater than the size of charptr1.

Table H.7. Summary of the results from test case 10

Warnings					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memmove	0x001017cb	charptr1	8	11
1	memmove	0x001017cb	charptr2	11	11
1	memmove	0x001017cb	int	12	11
2	memmove	0x00101880	charptr1	8	11
2	memmove	0x00101880	charptr2	11	11
2	memmove	0x00101880	int	12	11

Test Case 10 produced two warnings when it should have produced one. This is the result of the source location being misidentified as a pointer. There is no simple solution for this case; it would require extensive analysis of pointer usage within functions to determine the actual source. Such analysis would be better suited for dynamic analysis because the values at the pointers would need to be checked when they are used. This is needed because the values at the pointers can be different depending on the control flow that the program took.

Test 11 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *strncpy()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.8. Summary of the results from test case 11

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strncpy	0x001017aa	charptr1	10	11
1	strncpy	0x001017aa	charptr2	11	11
1	strncpy	0x001017aa	int	12	11

Test 12 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *wcsncpy()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 13 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memcpy()*
- Flow variant: Passes source values between files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 14 Results:

- Functional variant: CWE 193 - refers to an off by one error
- Sink: *memmove()*
- Flow variant: Void pointer to data passed from one function to another in different source files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 15 Results:

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *strncat()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
  - Assembly good2 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.
  - Assembly good2 function: Produced no messages.

Table H.9. Summary of the results from test case 15

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strncat	0x001017aa	charptr1	64	100
1	strncat	0x001017aa	charptr2	99	99
1	strncat	0x001017aa	int	100	100

Test 16 Results:

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *wcsncat()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.10. Summary of the results from test case 16

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcsncat	0x001017c8	charptr1	208	400
1	wcsncat	0x001017c8	charptr2	396	396
1	wcsncat	0x001017c8	int	400	400

Test 17 Results:

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *memmove()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.11. Summary of the results from test case 17

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memmove	0x001017bc	charptr1	64	100
1	memmove	0x001017bc	charptr2	99	99
1	memmove	0x001017bc	int	100	100

Test 18 Results:

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *memcpy()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 19 Results:

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *strncpy()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 20 Results:

- Functional variant: CWE 805 - refers to buffer access with incorrect length value.
- Sink: *wcsncpy()*
- Flow variant: Data passed as an argument from one function to a function in a different source file called via a function pointer
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 21 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *wcsncpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
  - Assembly good2 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.
  - Assembly good2 function: Produced one caution message (Msg 3). The size for charptr2 and int are greater than the size of charptr1.

Table H.12. Summary of the results from test case 21

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcsncpy	0x001017ff	charptr1	196	396
1	wcsncpy	0x001017ff	charptr2	408	396
1	wcsncpy	0x001017ff	int	408	396
Cautions					
2	wcsncpy	0x001018f3	charptr1	196	196
2	wcsncpy	0x001018f3	charptr2	408	196
2	wcsncpy	0x001018f3	int	408	196
3	wcsncpy	0x001019d9	charptr1	196	196
3	wcsncpy	0x001019d9	charptr2	408	196
3	wcsncpy	0x001019d9	int	408	196

Test 22 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *strncpy()*
- Flow variant: Uses conditional to control the flow of the program and passes source information between functions.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
  - Assembly good2 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.
  - Assembly good2 function: Produced one caution message (Msg 3). The size for charptr2 and int are greater than the size of charptr1.

Table H.13. Summary of the results from test case 22

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strncpy	0x00101801	charptr1	50	99
1	strncpy	0x00101801	charptr2	104	99
1	strncpy	0x00101801	int	104	99
Cautions					
2	strncpy	0x001018e3	charptr1	50	49
2	strncpy	0x001018e3	charptr2	104	49
2	strncpy	0x001018e3	int	104	49
3	strncpy	0x001019c5	charptr1	50	49
3	strncpy	0x001019c5	charptr2	104	49
3	strncpy	0x001019c5	int	104	49

Test 23 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *memcpy()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.

Table H.14. Summary of the results from test case 23

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memcpy	0x00101845	charptr1	196	396
1	memcpy	0x00101845	charptr2	408	396
1	memcpy	0x00101845	int	408	396
Caution					
2	memcpy	0x00101971	charptr1	196	196
2	memcpy	0x00101971	charptr2	408	196
2	memcpy	0x00101971	int	408	196

Test 24 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *wcsncat()*
- Flow variant: Uses a union for the source parameter.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.

Table H.15. Summary of the results from test case 24

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcsncat	0x0010180d	charptr1	196	396
1	wcsncat	0x0010180d	charptr2	408	396
1	wcsncat	0x0010180d	int	408	396
Caution					
2	wcsncat	0x00101901	charptr1	196	196
2	wcsncat	0x00101901	charptr2	408	196
2	wcsncat	0x00101901	int	408	196

Test 25 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *strncpy()*
- Flow variant: Passes source information between functions.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.

Table H.16. Summary of the results from test case 25

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strncat	0x0010182a	charptr1	50	99
1	strncat	0x0010182a	charptr2	104	99
1	strncat	0x0010182a	int	104	99
Caution					
2	strncat	0x00101935	charptr1	50	49
2	strncat	0x00101935	charptr2	104	49
2	strncat	0x00101935	int	104	49

Test 26 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *memmove()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.

Table H.17. Summary of the results from test case 26

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memmove	0x001017bc	charptr1	50	99
1	memmove	0x001017bc	charptr2	104	99
1	memmove	0x001017bc	int	104	99
Caution					
2	memmove	0x00101856	charptr1	50	49
2	memmove	0x00101856	charptr2	104	49
2	memmove	0x00101856	int	104	49

Test 27 Results:

- Functional variant: CWE 806 - refers to buffer access using size of source buffer
- Sink: *memcpy()*
- Flow variant: Data passed in an array from one function to another in different source files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 and int are greater than the size of charptr1.

Table H.18. Summary of the results from test case 27

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	memcpy	0x00101944	charptr1	50	99
1	memcpy	0x00101944	charptr2	104	99
1	memcpy	0x00101944	int	104	99
Caution					
2	memcpy	0x001019ea	charptr1	50	49
2	memcpy	0x001019ea	charptr2	104	49
2	memcpy	0x001019ea	int	104	49

Test 28 Results:

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for argument charptr2 is greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.19. Summary of the results from test case 28

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcat	0x001017b7	charptr1	64	99
1	strcat	0x001017b7	charptr2	99	99

Test 29 Results:

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *wcscat()*
- Flow variant: Uses pointers to point to the source values.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for argument charptr2 is greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.20. Summary of the results from test case 29

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcscat	0x001017c3	charptr1	208	396
1	wcscat	0x001017c3	charptr2	396	396

Test 30 Results:

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *wcscpy()*
- Flow variant: Uses a union for the source parameter.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill and Amount are greater than Dst.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for argument charptr2 is greater than the size of charptr1.
  - Assembly good1 function: Produced no messages.

Table H.21. Summary of the results from test case 30

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcscpy	0x001017c3	charptr1	208	396
1	wcscpy	0x001017c3	charptr2	396	396

Test 31 Results:

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *strcpy()*
- Flow variant: Passes source information between functions.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 32 Results:

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *wcscpy()*
- Flow variant: Passes source values between files.
- Character type: wide char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 33 Results:

- Functional variant: Destination parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Data passed in a struct from one function to another in different source files.
- Character type: char
- Expected result:
  - Assembly bad function: No message. No overflow.
  - Assembly good1 function: No message. No overflow.
- Actual results:
  - Assembly bad function: Produced no messages.
  - Assembly good1 function: Produced no messages.

Test 34 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *wcscpy()*
- Flow variant: Uses conditional to control the flow of the program.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.22. Summary of the results from test case 34

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcscpy	0x001017d0	charptr1	208	396
1	wcscpy	0x001017d0	charptr2	408	396
Caution					
2	wcscpy	0x0010188d	charptr1	208	196
2	wcscpy	0x0010188d	charptr2	408	196

Test 35 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *wcscat()*
- Flow variant: Uses conditional to control the flow of the program and passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
  - Assembly good2 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.
  - Assembly good2 function: Produced one caution message (Msg 3). The size for charptr2 is greater than the size of charptr1.

Table H.23. Summary of the results from test case 35

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcscat	0x00101805	charptr1	208	396
1	wcscat	0x00101805	charptr2	408	396
Cautions					
2	wcscat	0x00101905	charptr1	208	196
2	wcscat	0x00101905	charptr2	408	196
3	wcscat	0x001019f7	charptr1	208	196
3	wcscat	0x001019f7	charptr2	408	196

Test 36 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcpy()*
- Flow variant: Uses pointers to point to the source values.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.24. Summary of the results from test case 36

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcpy	0x00101835	charptr1	64	99
1	strcpy	0x00101835	charptr2	104	99
Caution					
2	strcpy	0x00101957	charptr1	64	49
2	strcpy	0x00101957	charptr2	104	49

Test 37 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Uses a union for the source parameter.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.25. Summary of the results from test case 37

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcat	0x00101802	charptr1	64	99
1	strcat	0x00101802	charptr2	104	99
Caution					
2	strcat	0x001018f1	charptr1	64	49
2	strcat	0x001018f1	charptr2	104	49

Test 38 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *wcscat()*
- Flow variant: Passes source information between functions.
- Character type: wide char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.26. Summary of the results from test case 38

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	wcscat	0x001017f1	charptr1	208	396
1	wcscat	0x001017f1	charptr2	408	396
Caution					
2	wcscat	0x001018cf	charptr1	208	196
2	wcscat	0x001018cf	charptr2	408	196

Test 39 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcat()*
- Flow variant: Passes source values between files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.27. Summary of the results from test case 39

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcat	0x0010179d	charptr1	50	99
1	strcat	0x0010179d	charptr2	104	99
Caution					
2	strcat	0x00101824	charptr1	50	49
2	strcat	0x00101824	charptr2	104	49

Test 40 Results:

- Functional variant: Source parameter passed to sink as a pointer
- Sink: *strcpy()*
- Flow variant: Data passed as a global variable from one function to another in different source files.
- Character type: char
- Expected result:
  - Assembly bad function: Produces one warning message. Src fill is greater than Dst.
  - Assembly good1 function: Produces one caution message. Src size is greater than Dst.
- Actual results:
  - Assembly bad function: Produced one warning message (Msg 1). The max fill for arguments charptr2 and int are greater than the size of charptr1.
  - Assembly good1 function: Produced one caution message (Msg 2). The size for charptr2 is greater than the size of charptr1.

Table H.28. Summary of the results from test case 40

Warning					
Msg	Sink	Sink Location	Argument	Size	Max Fill
1	strcpy	0x001018f8	charptr1	50	99
1	strcpy	0x001018f8	charptr2	104	99
Caution					
2	strcpy	0x00101986	charptr1	50	49
2	strcpy	0x00101986	charptr2	104	49

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] NSA, "Ghidra," 2019. Available: https://www.nsa.gov/resources/everyone/ghidra/

[2] NSA, "Ghidra API," http://ghidra.re/ghidra_docs/api/, 2020.

[3] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, vol. 45, no. 10, pp. 88–90, Oct 2012.

[4] H. Zhu, T. Dillig, and I. Dillig, "Automated inference of library specifications for source-sink property verification," in *Asian Symposium on Programming Languages and Systems*. Springer, 2013, pp. 290–306.

[5] C. ISO, "Programming languages—C," *American National Standards Institute, ISO/IEC*, vol. 9899, 1999.

[6] MITRE Corporation, "2019 CWE top 25 most dangerous software errors," 2019. Available: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

[7] NIST, "National vulnerability database." Available: https://nvd.nist.gov/

[8] MITRE, "Common vulnerabilities and exposures," https://cve.mitre.org/, May 2020.

[9] H. Booth, D. Rike, and G. Witte, "The national vulnerability database (nvd): Overview," National Institute of Standards and Technology, Tech. Rep., 2013.

[10] Y. Xie, A. Chou, and D. Engler, "Archer: using symbolic, path-sensitive analysis to detect memory access errors," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 327–336, 2003.

[11] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities." in *NDSS*, 2000, pp. 2000–02.

[12] G. Holzmann, "Static source code checking for user-defined properties," in *Proc. IDPT*, 2002, vol. 2.

[13] S. Flisakowski, "C-tree distribution," 1997. Available: https://github.com/nimble-code/Uno/blob/master/Src/tree.h

[14] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.

[15] R. Kindermann, "Static detection of buffer overflows in executables," 2008. Available: https://www.academia.edu/3859898/Static_Detection_of_Buffer_Overflows_in_Executables_Diplomarbeit

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[17] Q. Meng, C. Feng, B. Zhang, and C. Tang, "Assisting in auditing of buffer overflow vulnerabilities via machine learning," *Mathematical Problems in Engineering*, vol. 2017, 2017.

[18] B. M. Padmanabhuni and H. B. K. Tan, "Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, July 2015, vol. 2, pp. 450–459.

[19] B. M. Padmanabhuni and H. B. K. Tan, "Predicting buffer overflow vulnerabilities through mining light-weight static code attributes," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov 2014, pp. 317–322.

[20] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. 7, pp. 65 889–65 912, 2019.

[21] B. Knighton and C. Delikat, "Black Hat USA 2019," Aug 2019. Available: https://github.com/NationalSecurityAgency/ghidra/wiki/files/blackhat2019.pdf

[22] P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[23] MITRE, CWE, "Common weakness enumeration," 2006. Available: http://cwe.mitre.org

[24] MITRE, CWE, "CWE-121: Stack-based buffer overflow," 2020. Available: https://cwe.mitre.org/data/definitions/121.html

[25] *strcpy(3) Linux Programmer's Manual*, October 2019. Available: https://linux.die.net/man/3/strcpy

[26] *printf(3) Linux Programmer's Manual*, October 2019. Available: http://man7.org/linux/man-pages/man3/printf.3.html

[27] *scanf(3) Linux Programmer's Manual*, October 2019. Available: http://man7.org/linux/man-pages/man3/alloca.3.html

[28] *alloca(3) Linux Programmer's Manual*, March 2019. Available: http://man7.org/linux/man-pages/man3/alloca.3.html

[29] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009. Available: https://doi.org/10.1145/1609956.1609960

[30] G. Michaelson, "Bernhard Steffen, Oliver R¨uthing, and Michael Huth: Mathematical foundations of advanced informatics—volume 1: Inductive approaches," *Formal Aspects of Computing*, vol. 31, no. 5, pp. 641–642, Nov 2019. Available: https://doi.org/10.1007/s00165-019-00496-x

[31] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," *NIST special Publication*, vol. 800, no. 142, p. 142, 2010.

[32] GCC Team *et al.*, "GCC, the GNU Compiler Collection-GNU Project-Free Software Foundation (FSF)," *GCC, the GNU Compiler Collection*.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California