ODSS: A Ghidra-based Static Analysis Tool for Detecting Stack-Based Buffer Overflows

Eric Wikman
United States Naval Academy
wikman@usna.edu

Thuy D.Nguyen
Naval Postgraduate School
tdnguyen@nps.edu

Cynthia Irvine
Naval Postgraduate School
irvine@nps.edu

Abstract

To reduce code exploitabilty, techniques for analyzing binaries for potential buffer overflow vulnerabilities are needed. One method is static analysis, which involves inspection of disassembled binaries to identify exploitable weaknesses in the program. Buffer overflows can occur in libc functions. Such functions can be referred to as vulnerable sinks. We present Overflow Detection from Sinks and Sources (ODSS), a script written for the Ghidra API to search for vulnerable sinks in a binary and to find the source of all the parameters used in each sink. We conduct static analysis of ten common libc functions using ODSS, and show that it is possible to both find overflow vulnerabilities associated with functions using stack-allocated strings and to determine the feasibility of a buffer overflow exploitation.

Key words: *binary analysis, buffer overflow, Ghidra, libc functions, static analysis*

1. Introduction

Although buffer overflow vulnerabilities have been documented for five decades [1], they remain a serious cybersecurity problem. Substantial effort has been expended to develop techniques to find flaws in program binaries [2], and researchers have systematized a range of binary analysis techniques [3]. Yet, despite their "ancient" heritage, new buffer overflow vulnerabilities emerge with great regularity, as recorded by MITRE [4].

Among the techniques used to exploit buffer overflow vulnerabilities is to pass bad parameters to vulnerable sinks, where 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 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. Tools are needed to assist in software reverse engineering (SRE) aimed at identifying buffer overflows.

In this paper, we present Overflow Detection from Sinks and Sources (ODSS) [5], a static analysis technique to detect stack-based buffer overflow vulnerabilities in binaries compiled for execution on Linux x86 64-bit systems. ODSS has been implemented as a Python script that uses the Ghidra API [6] to analyze binaries for buffer overflow vulnerabilities. The Juliet test suite [7] is used to evaluate ODSS's effectiveness.

Our contributions are:

- ODSS static analysis tool to discover buffer overflow vulnerabities using sinks and sources,
- a Python implementation of ODSS, and
- demonstration of the tool's identification of buffer overflow vulnerabilities in ten selected *libc* functions.

To explain ODSS, we provide background and related work in Section 2. Our design approach is presented in Section 3. Functional testing of the ODSS tool is described in Section 4. A discussion and conclusion follow in Section 5.

2. Background

When a buffer allocated on the stack is indexed outside of its defined boundaries, the location provided is said to be out of bounds and a stack-based buffer overflow condition occurs. This is common in languages such as C and C++. Usually, these errors are caused by improper input validation within the program. Good input validation checks the type and size of an input before using it. When these checks are absent, a program may fail or be vulnerable to exploitation. This can occur in functions such as *strcpy()*, where the source string is copied into the destination string. *Strcpy()* does not check the size of its input strings and instead uses the null terminator in the source string to know when to stop copying. If the source string is larger than the destination string, *strcpy()* will write past the end of the destination string, causing a buffer overflow.

2.1. Ghidra

Ghidra [8] is a SRE framework developed by the National Security Agencey (NSA) that runs on Linux, OS X, and Windows. It supports SRE of executables for a wide range of processors. Among its features, Ghidra assists in identifying items on the stack and finding object references (e.g., address locations, function locations, and stack locations) within a binary file [9].

Ghidra supports the analysis of disassembled code [10]. It 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*). Search of a Ghidra-generated function database allows analysts to easily identify *libc* functions known to be vulnerable to buffer overflows.

Approximating the memory consumed by local variables plays a large role in determining the stack space allocated to the function. Without Ghidra, a user must inspect the disassembled function and determine the amount of stack space allocated to local variables. This is typically accomplished by looking at the beginning of the function for the `SUB RSP, 0x**` instruction. However, there are no indications of where one local variable starts and another ends – the analyst must laboriously map all of a function's stack offsets. In contrast, Ghidra performs this analysis, thus allowing 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 amount stack space allotted to that reference.

2.2. Juliet Test Suite

The Juliet test suite was developed by NSA and published by NIST [7]. The suite is described as, "... a systematic set of thousands of small test programs in C/C++ and Java, exhibiting over 100 classes of errors, ... " [11]. The latest release is comprised of over 86,000 programs, each containing known flaws. Items in the Juliet suite are categorized according to Common Weakness Enumerations (CWEs) [12]. In this work, we use CWE 121 test cases, stack-based buffer overflows.

2.3. Related Work

Recently, companies have developed methods, often proprietary, for buffer overflow vulnerability discovery. However, some open research describes methods specifically for uncovering buffer overflow vulnerabilities.

ARCHER is a constraint solver that checks the bounds of variables and memory sizes of various objects. ARCHER statically analyzes source code, not binaries, for memory constraint violations such as array accesses, pointer dereferences, or calls to a function that expects a size parameter [13].

BOON detects buffer overflows in source code by using integer range analysis [14]. The target program is parsed to find string variables, which are assigned two integers: the string's allocated size and the number of bytes currently in use. BOON checks the usage of each string to determine if its length is greater than the allocated size. ODSS uses this comparison technique.

To reduce the false positive rate in software vulnerability discovery, Holzmann developed UNO [15]. UNO checks for common errors, e.g., uninitialized variables, nil-pointer dereferencing, and out-of-bound array indexing. In addition, users can customize UNO to check for flaws that may be uncommon in general, but of interest in the case of a particular application. UNO requires two passes through the code. In the first pass, UNO builds a parse tree using ctree [16]. It then converts the tree into a control flow graph. In the second pass, UNO performs buffer overflow analysis on the control flow graph. It checks for errors by performing a global analysis based on its first pass [15]. Unlike Archer, BOON, and UNO, ODSS looks at disassembly information from binaries as opposed to the source code.

Value set analysis recovers "information about the contents of machine registers and memory locations at every program point in an executable" [17]. This technique was used by Kindermann [18] to perform buffer overflow detection via the static analysis of executables. Kindermann's buffer overflow detection method focused on identifying buffer overflows caused by loops. Our approach focuses on finding the source parameters for a given sink. Once the source parameters are found, our method does perform a limited version of this type of value set analysis to determine if the sink can be overflowed.

Machine learning has also been used to predict buffer overflows from vulnerable sinks. Common approaches use supervised learning and neural networks to statistically classify overflows. Elements that contribute to sinks are classified using various machine learning algorithms. The goal is to determine the probability that a particular sink will cause an overflow. This machine learning method has been used on source code as well as binary files [19, 20, 21, 22]. These techniques are probabilistic; further analysis of the identified vulnerabilities is required to determine if they are both valid and exploitable. ODSS can support such further analysis.

3. Design Approach

We explore buffer overflow vulnerability detection in *libc* functions through static analysis. The *libc* functions capable of producing a buffer overflow are referred to as *vulnerable sinks*. Here, a *sink* is a function that receives inputs to execute its code. This section covers how sinks can be found and introduces 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 memory location within the process namespace [23]. When a source is a value, it may be stored in a register before a function call. When it is a variable, the source may be located on the stack or heap. With ODSS, we developed a new approach to discover buffer overflows from vulnerable sinks by tracing the sink's parameters back to their sources. ODSS allows analysts to determine if there is a mismatch in parameter sizes that would cause an overflow. Here, we describe the automation of the ODSS process.

3.1. Primary Causes of Buffer Overflows

Buffer overflows often occur due to improper size validations and missing or incorrect input validation for a given buffer. These can occur when loop indicies are incremented past the bounds of an array, or when functions indiscriminately move values into buffers. Common *libc* functions such as *strcpy()*, *strcat()*, *memmove()*, and *fgets()* are known to be susceptible to buffer overflow. These functions do not validate their input parameters, and they are sinks because they receive their values and variables from sources external to the function. This does not mean that all sinks can produce a buffer overflow; however, failure to check its parameters renders a function less prepared to handle abnormal input. Consider *strcpy()*: it takes two parameters, (1) a destination and (2) a source string. Character-by-character, *strcpy()* reads from the source string and writes to the destination string. Once a null byte is read, the null byte is written to the destination string and the function returns [24]. *Strcpy()* has no internal mechanism for checking if the destination buffer is large enough to contain the source string – the programmer must perform those checks prior to calling *strcpy()*. If the source string is larger than the destination buffer, then the function will write past the end of the destination buffer, causing an overflow. Vulnerable sinks are common entry points for attackers to cause buffer overflows [25]. Thus, performing ODSS analysis on vulnerable *libc* 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 in order to perform bounds checking. Although each performs bounds checking differently, they base their approaches on the idea that every variable has space allocated to it and can be filled (initialized) with a string. Bounds checking then compares the allocated size and fill amounts of two variables to see if a buffer overflow is possible. The overflow occurs when the source variable's fill amount is larger than the allocated size of the destination variable. If a source variable's allocated size is larger than the destination's allocated size, an overflow is not guaranteed, but this does indicate a possible problem.

When using source code, finding parameters' sources is less challenging. Each parameter name resolves to some initialized variable or input parameter for a given function. Initialized variables are clearly defined with the size of the data construct (e.g., array or string) used to initialize the variable. Changes to a variable can be found by searching for the variable's name, then determining how the variable is used in the function. This task is more complex when sources are passed as parameters. To determine the location of each possible source one must find all the references to the called function and check the variable or value used as a parameter by each calling function. Similarly, sinks and their parameters can be found by searching for each sink's name and parameters in the source code.

For the disassembly generated from binary code, most of the sources and sinks become addresses. When the sources are local variables, they become offsets relative to the address of the current stack frame. Given a single stack offset, it is difficult to determine how much stack space is allocated to a particular local variable. To determine a variable's allocated size, we need the start address of the preceding local variable. The allocated size of the source variable is the difference between the two stack locations. 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

ODSS automates the daunting manual process used to discover buffer overflows in binary code using sinks. Typically, vulnerable sinks could be called hundreds of times. Automating this process can significantly reduce the time required to analyze vulnerable sinks.

At a high-level, we summarize the ODSS approach in four steps. STEP 1: Identify the `CALL` instructions used to call the *libc* sinks within the binary file. STEP 2: Identify the source location for all of the parameters used in the sink call. STEP 3: Determine how the sources are used. STEP 4: Calculate if any combination of sources could overflow a buffer for a given sink.

STEP 1: Find Sinks. Vulnerable sinks can be found 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. Thus, all sinks used in the program can be identified. Ghidra creates a reference database, which contains reference information about functions, global variables, initialized data section(s), and the uninitialized data section. The reference database tracks the addresses from which each function is called. The references to the sinks contain the parameters passed to each sink. Thus, the references to the sinks become the starting point to search for the sources.

STEP 2: Find Sources. On 64-bit x86 systems, the first six parameters to functions are passsed via registers, while any additional parameters are passed on the stack. This requires the detection method to track register usage and stack locations from calling functions. Finding sources starts with the address where the sink is called. Since the *libc* functions that are chosen as sinks in this research have fewer than six parameters, all parameters are stored in registers.

By tracking register usage backwards through the code, it is possible to determine values that are loaded into the registers. These values passed by the calling function can include static values, local variables, function parameters, or addresses. For static values, local variables, and addresses, the search ends because these values indicate a location or size; 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. All the references to the function parameters or pointers must be checked. For each function parameter, a path tree is formed where the call to the sink is the root and the functions containing the sources are the leaves and interior nodes. For the tree to be built properly, repeat functions cannot appear on the same path. This means that ODSS does not double search functions, to prevent loops in the tree.

Figure 1 shows a sink that can have multiple sources for its parameters. The sink takes two parameters: `par1` and `par2`. For clarity, the parameter names at the sink are propagated with the same name back up the call tree to the leaves. Both of the sink's parameters come from the parameters of *Func1()*. *Func2()* and *Func3()* call *Func1()*, so those functions are added to the tree. *Func2()* is similar to *Func1()* because both of the parameters in the call to *Func1()* come from *Func2()*'s parameters. In *Func3()*, `par2` is allocated inside the function. For *Func4()*, *Func5()*, and *Func6()*, `par1` is allocated in each of these functions.

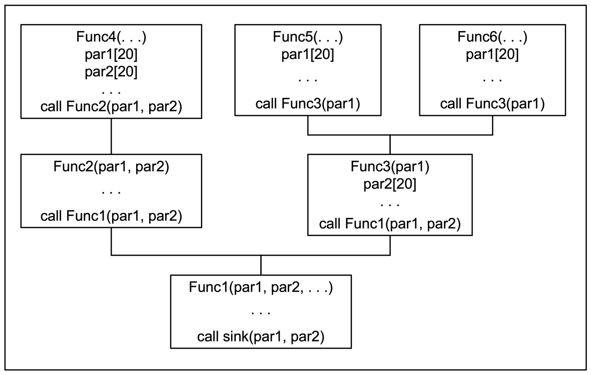
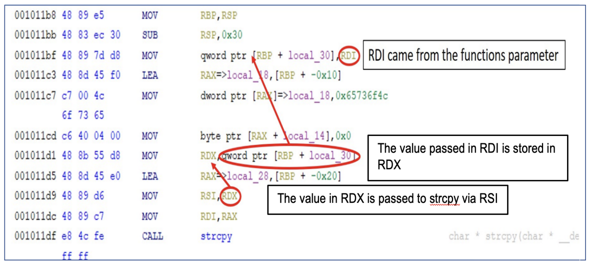


Figure 1. Source tree with the sink as the root

Figure 2 gives a simple example of the ODSS tracking technique. The disassembly shown in the figure ends with a call to *strcpy()*. Register `RSI`, the second parameter to *strcpy()*, is the sink parameter of interest. At offset `001011d9`, `MOV RSI, RDX` shows that `RDX` holds the value that must be tracked. At offset `001011d1`, we find `MOV RDX, qword ptr [RBP + local 30]`. The `qword ptr [RBP + local 30]` operand is a local variable of this function. At offset `001011bf`, `qword ptr [RBP + local 30]` is loaded with `RDI`. We see that, at this point, `RDI` was the first parameter passed to the function. Hence, the second parameter to *strcpy()* came from the function's parameter. This means that, to find each source of the second parameter to the sink, all references to each source in the call chain must be checked.



```
001011b8 48 89 e5       MOV       RBP,RSP
001011bb 48 83 ec 30    SUB       RSP,0x30
001011bf 48 89 7d d8    MOV       qword ptr [RBP + local_30],RDI
001011c3 48 8d 45 f0    LEA       RAX=>local_18,[RBP + -0x10]
001011c7 c7 00 4c       MOV       dword ptr [RAX]=>local_18,0x65736f4c
         6f 73 65
001011cd c6 40 04 00    MOV       byte ptr [RAX + local_14],0x0
001011d1 48 8b 55 d8    MOV       RDX,qword ptr [RBP + local_30]
001011d5 48 8d 45 e0    LEA       RAX=>local_28,[RBP + -0x20]
001011d9 48 89 d6       MOV       RSI,RDX
001011dc 48 89 c7       MOV       RDI,RAX
001011df e8 4c fe       CALL      strcpy                          char * strcpy(char * __de
         ff ff
```

RDI came from the functions parameter

The value passed in RDI is stored in RDX

The value in RDX is passed to strcpy via RSI

Figure 2. Ghidra disassembly of a source being passed to a sink

STEP 3: Determine Source Usage. Prior to calculating the overflows for a sink, how the source parameter may have been used or modified by intermediate functions prior to its use by the sink needs to be determined. This requires 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 where the source was found. From there, the source is tracked through the call tree to determine how it was used enroute to the sink. Within every function where the source is used as a parameter, the source is checked to determine if it was used in a call to another function. If so, then the new function is added to the list of functions to be checked. To prevent a recursive loop, functions are checked once. Source usage is determined in two passes. The first pass collects values that have no 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.

STEP 4: Calculate Overflow. To calculate overflows, all sources for a given sink are compared to determine if a buffer overflow is possible. A common format is used to record parameter attributes for each sink: a source string, a destination string, and sometimes an integer byte count. The source string specifies where the values came from and the destination string specifies where the values will be written. Using the attributes 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 of all three parameters are checked to determine if an overflow might occur. This method of calculating overflow has three resulting cases:

- Safe. There is no overflow. Nothing is reported.
- Caution. The allocated size of the source is larger than the allocated size of the destination; however, the maximum fill amount of the source is less than the allocated size of the destination.
- Warning. The source's string length is greater than the destination's allocated size.

Overall Complexity With an understanding of the steps for finding overflows, we can determine the complexity as followed:

- n = number of functions in the program
- m = number of calls to a given vulnerable sink
- l = lines of code in the program
- s = number of sources for a given call to a sink

In STEP 1, finding the sinks in Ghidra's function database is a linear lookup, where n is the number of functions in the program. In STEP 2, finding the sources for a given call to the sink requires searching up through the code: for every m there is some number of s. In the worst case of searching for all the sources, it could search through every line of code in the program. However, for this step, dynamic programming is used to limit searching for sources that belong to multiple sinks. This is determined by examining the path a source takes between functions and comparing against other paths that took the identical route. Even with the dynamic programming in STEP 2, this would still be calculated as msl. Finding source usage in STEP 3 takes two passes through the code for every source. Again, in the worst case, this would require searching through every line of code for every source. Dynamic programming is applied again so that duplicate paths for a given source can be avoided. This will still result in $m2(sl)$. In STEP 4, calculating the overflow involves comparing the sources that contain the values against the sources that will receive the values. In the case of the sink involving three sources, this would be ms^3. Combining these four steps results in:

$$O(n + m(3sl + s^3))$$

Depending on the program, any one of these variables could be the dominant feature of the run time of the overflow search tool.

3.4. Core Modules of the ODSS Tool

The ODSS software consists of three modules: *Main, Sink*, and *Source*. The Main module is responsible for the program's execution flow. The Sink module finds, creates, and calculates the overflows for the sinks. The Source module is responsible for finding, creating, and filling in the attributes for sources.

The Sink module encapsulates[1] attributes for all of the vulnerable *libc* functions calls in the binary. Its processing includes discovering the sinks, creating a database entry for each sink, and determining whether a particular sink can overflow.

The Source module finds the sources of the parameters that will be passed to the vulnerable *libc* functions. It also determines how the parameter will be used. To find the sources, the sink's parameters must be traced back to their 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 is determined. Searching for the usage of each source yields the longest string length used in the source. This requires checking if the source is modified along its path to the sink. Knowing the longest string length used in the source allows its string length to be compared with allocated

[1]We refer to databases in the manner described by Parnas [26]

sizes of other sources to determine if a buffer can be overflowed.

3.5. Design Choices

This section discusses the design choices for ODSS.

Ghidra Use: The Ghidra API supports finding functions, references, and iterating through instructions [6]. Ghidra's *function finding* API supports the identification of sinks. Ghidra's *reference finding* API provides 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 through the binary file significantly easier. Ghidra's active support, open availability, and function database facilitated the ODSS implementation. These criteria led to its choice over several other SRE tools such as IdaPro [27], the advanced functionality for which is proprietary; Cerbero, which is proprietary [28]; and API Monitor [29], which has not been maintained recently. This work utilized Linux-based Ghidra version 9.1-BETA and its associated API.

Linux x86 64-bit system: ODSS is designed to run on a Linux x86 64-bit operating system. As noted in Section 3.3, the first six parameters are passed in registers. Passing parameters in registers presents a unique, but solvable, challenge to tracking how values are moved between functions.

Buffer overflow detection used on C programs: Since ODSS is based on vulnerabilities in *libc* functions, the script only supports C programs. Focusing on a single programming language results in consistency across the different tests and allows uniform analysis.

Compilation requirements: All test programs were compiled using the default GCC configurations. The only exception is the *-fno-builtin* option, which prevents functions from being inlined. In this context, an inlined function is placed directly into the code instead of using `CALL` to invoke the function. Certain *libc* functions, such as *memmove()*, are inlined when compiled using the default compiler settings. Test cases with multiple files are statically compiled together.

Buffer overflows in stack-allocated strings: We focus only on the overflows that occur from stack-allocated strings, which are better suited for static analysis. Sizes of the variables allocated on the stack are known before run time. The sizes of variables allocated on the heap can change during program execution. For heap-based variables, program flow must be determined to correctly calculate the space allocated to each variable, which is not conducive to static analysis methods.

Functional testing using the Juliet test suite: The set of vulnerabilities from the Juliet test suite [7] relevant to this work comes from the section labeled *CWE 121 stack-based buffer overflow*. These vulnerabilities provide numerous examples of the type of buffer overflows ODSS is intended to detect.

3.6. Restrictions and Exclusions

Several functional features and capabilities are beyond the scope of this work.

Variadic functions: Variadic functions take a variable number of parameters. Examples from *libc* are *printf()* and *scanf()* [24]. A format specifier, which is passed as the first parameter, is used to parse the remaining parameters. Variadic functions are excluded from this work. A dynamic approach that parsed the format specifier would be needed.

Stack manipulation functions: Some functions adjust the stack frame. An example is *alloca()* [24]. Since the *alloca()* function is inlined to the function that calls it, it is difficult to quickly find *alloca()* when searching through the program, and thus transforms the task of determining if a buffer overflow could occur into one of searching for code that exhibits *alloca()* behavior. Hence, *alloca()* and similar functions are not included in the buffer overflow tests. Our method relies on Ghidra's function database to identify calls to functions. Future work in this area involves the identification of these inlined functions. This feature is not an organic part of Ghidra's functionality.

Values based on complex algorithms: Complications arise when a program uses more than basic arithmetic to calculate values. The equation used to calculate those values must be discovered to determine what parameters are sent to the sinks. To determine values resulting from complex equations, a state machine that modeled those equations would be required. Our work is intended to explore the feasibility of ODSS, so development of a state machine model was left to future work. Instead, we used addition and multiplication equations to determine input values to the sinks, and excluded the use of instructions such as `SUB` or `DIV` when calculating sizes.

Flow invariant: Often, programs consist of many conditionals that control the code's execution path. From a static analysis perspective, a control flow graph [30] could be constructed to determine how a program executes. The goal of our overflow detection method is to test the feasibility of tracking a source back to its origin. To constrain the scope of our experiments, flow control is not analyzed. Knowing the program's control flow can increase the accuracy of the detection method;

however, mapping the control flow of a program is a separate large topic, e.g., [31].

Multiple Register tracking: ODSS assumes that only one register is used to calculate a source's location. For the Juliet test cases, tracking one register was adequate. More complex programs use different memory addressing modes that require using multiple registers to calculate a source's true location. This is common when indexing into an array; 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 used in an instruction. Once the values for the registers are found, those values could be used to calculate the source's location. This enhancement is future work.

Limited detection of overflow from concatenation functions: Since program control flow is not tracked, knowing when a source has values concatenated to the source is not considered. A single-use overflow would copy 100 bytes into a 50-byte buffer once. A continuous overflow would copy 1 byte into a 50-byte buffer 100 times. We focused on single-use overflows.

4. Functional Tests

We conducted functional tests of ODSS on ten different sinks with a total of 40 tests: *strcpy(), strncpy(), memcpy(), strcat(), strncat(), wcscat(), wcsncat(), strcat(), wcscpy(),* and *wcsncpy()*. The functional tests follow NIST combinatorial testing guidance [32]. By categorizing the different tests, we were able to perform pairwise combinations on the test cases to ensure coverage of the available problem set. This involved grouping sinks with a buffer overflow variant. We then grouped the test cases based on how the sources got to the sink. This is referred to as the flow variant. The flow variants are based on the flow type (control flow or data flow) and upon the way parameters are declared. We then ensured there was a test case for every overflow variant interaction with every possible flow variant, as listed in Table 1. This allows us to reduce the total number of test cases to 40 tests.

Sinks were divided into two groups: two- and three-parameter sinks. In the cases of a three-parameter sink, we also needed to consider the integer byte count to determine an overflow. Because of this distinction, there are multiple tests for the applicable overflow/flow variant combination. Wide and normal character types can be grouped together because the source's size is determined by counting the bytes at the location of the source; differences between wide or normal characters will not skew the tests.

Table 1. Number of test cases per interaction between overflow variants (OV) and flow variants (FV). The total is 40 cases.

	FV1	FV2	FV3	FV4	FV5	FV6	FV7
OV1	2	N/A	N/A	N/A	N/A	N/A	N/A
OV2	2	N/A	2	2	2	2	2
OV3	1	N/A	1	1	1	1	1
OV4	1	1	1	1	1	1	1
OV5	1	N/A	1	1	1	1	1
OV6	1	1	1	1	1	1	1

*N/A indicates there was no applicable test case for the flow variant and overflow variant combination.

FV1: Uses a conditional to control the flow of the program.

FV2: Uses a conditional to control the flow of the program and passes source information between functions.

FV3: Uses pointers to point to the source values.

FV4: Uses a union for the source parameter.

FV5: Passes source information between functions.

FV6: Passes source values between files.

FV7: Pass values to different functions, but pass different types of information.

OV1: Incorrect length value used for a struct (CWE 121). *memmove(), memcpy()*

OV2: Off by one error (CWE 193). *strcpy(3), wcscpy(3), memcpy(2),memmove(2), strncpy(), wcsncpy()*

OV3: Buffer access with incorrect length value (CWE 805). *strncat(),wcsncat(),memmove(),memcpy(),strncpy(),wcsncpy()*

OV4: Buffer access using size of source buffer (CWE 806). *wcsncpy(), strncpy(2), memcpy(2), wcsncat(), memmove()*

OV5: Destination parameter passed to sink as a pointer (CWE 121). *strcat(2), wcscat(), wcscpy(2), strcpy()*

OV6: Source parameter passed to sink as a pointer (CWE 121). *wcscpy(), wcscat(2), strcpy(2), strcat(2)*

4.1. Test Results

Functional testing showed that tracking sources to sinks is a viable method for detecting overflows caused by vulnerable sinks. Our buffer overflow detection method correctly identified buffer overflows in 38 out of the 40 test cases. Figure 3 shows the receiver operating characteristic (ROC) curve against the 40 test cases, where ODSS produced only three false positives. In most test cases, the detection method accurately estimated the allocated sizes and fill sizes of the sources. The primary causes of deviations in the size estimates were due to accounting or not accounting for a null byte.

Two test cases failed, resulting in false positives. Test Case 2 checks the overflow variant using the *memmov()* sink. It tests a condition of struct overrun caused by an incorrect value used for a struct. Test Case 10 checks the overflow variant type using the *memmov()* sink. It tests an off-by-one error.

4.2. Review of error rates

First, Test Case 2 produced a warning for two good implementations of a sink. This is caused by

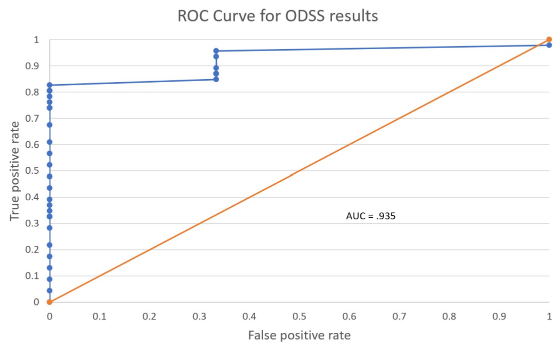
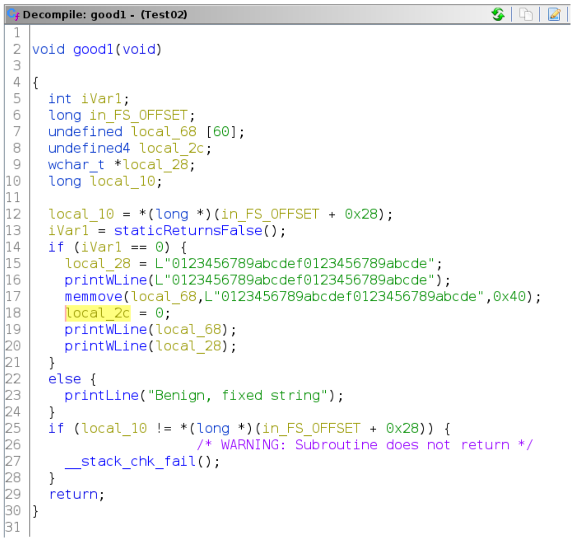


Figure 3. ODSS ROC curve

the destination parameter not accounting for the null byte at the end of the char array. Figure 4 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. 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 checks for the presence of a null bytes after a char array.



Decompile: good1 - (Test02)

```

void good1(void)

{
  int iVar1;
  long in_FS_OFFSET;
  undefined local_68 [60];
  undefined4 local_2c;
  wchar_t *local_28;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = staticReturnsFalse();
  if (iVar1 == 0) {
    local_28 = L"0123456789abcdef0123456789abcde";
    printWLine(L"0123456789abcdef0123456789abcde");
    memmove(local_68,L"0123456789abcdef0123456789abcde",0x40);
    local_2c = 0;
    printWLine(local_68);
    printWLine(local_28);
  }
  else {
    printLine("Benign, fixed string");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}

```

Figure 4. Error in Test Case 2 caused by null byte

Test Case 10 produced two warnings, but should have produced one (see Listing 1). This is because the source location was misidentified as a pointer. Figure 5 shows the disassembly that caused misidentification of 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.

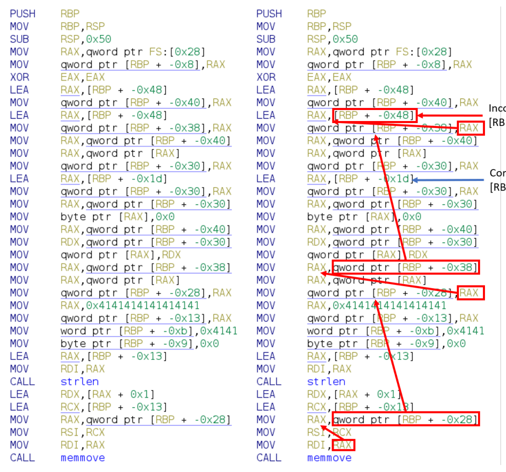
```
void ...CWE193_char_declare_memmove_32_bad()
{
    char * data;
    char * *dataPtr1 = &data;
    char * *dataPtr2 = &data;
    char dataBadBuffer[10];
    char dataGoodBuffer[10+1];
    {
        char * data = *dataPtr1;
        data = dataBadBuffer;
        data[0] = '\0';
        *dataPtr1 = data;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

Listing 1. Source code for Test Case 10

There is no simple solution for this case. Analysis of pointer usage within functions is needed to determine the source. Dynamic analysis is more appropriate since the values of the dereferenced pointers must be checked at the time of their use. Through the execution of a particular control flow path the target values are set.

5. Discussion and Conclusion

The tool focuses on function-to-function interactions, testing how sources pass through different functions enroute to 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



```
PUSH    RBP
MOV     RBP,RSP
SUB     RSP,0x50
MOV     RAX,qword ptr FS:[0x28]
MOV     qword ptr [RBP + -0x8],RAX
XOR     EAX,EAX
LEA     RAX,[RBP + -0x48]
MOV     qword ptr [RBP + -0x40],RAX
LEA     RAX,[RBP + -0x48]
MOV     qword ptr [RBP + -0x38],RAX
MOV     RAX,qword ptr [RBP + -0x40]
MOV     RAX,qword ptr [RAX]
MOV     qword ptr [RBP + -0x30],RAX
LEA     RAX,[RBP + -0x1d]
MOV     qword ptr [RBP + -0x30],RAX
MOV     RAX,qword ptr [RBP + -0x30]
MOV     byte ptr [RAX],0x0
MOV     RAX,qword ptr [RBP + -0x40]
MOV     RDX,qword ptr [RBP + -0x30]
MOV     qword ptr [RAX],RDX
MOV     RAX,qword ptr [RBP + -0x38]
MOV     RAX,qword ptr [RAX]
MOV     qword ptr [RBP + -0x28],RAX
MOV     RAX,0x4141414141414141
MOV     qword ptr [RBP + -0x13],RAX
MOV     word ptr [RBP + -0xb],0x4141
MOV     byte ptr [RBP + -0x9],0x0
LEA     RAX,[RBP + -0x13]
MOV     RDI,RAX
CALL    strlen
LEA     RDX,[RAX + 0x1]
LEA     RCX,[RBP + -0x13]
MOV     RAX,qword ptr [RBP + -0x28]
MOV     RSI,RCX
MOV     RDI,RAX
CALL    memmove
```

Figure 5. Incorrect source identification due to pointer use. (Left: original. Right: mark-up.)

showed that the tool needs a more robust means for tracking pointer usage to correctly identify sources. Other improvements include improving the accuracy of the source size calculations, properly identifying data types in uninitialized structs, and determining sources from multiple register usage.

```c
int main()
{
    char string1[50];
    char string2[50];

    string1[13] = 'A';

    for(int i = 0; i < 50; i++){
        if(i == 13){
            string2[i] = 'A';
        }
    }
}
```

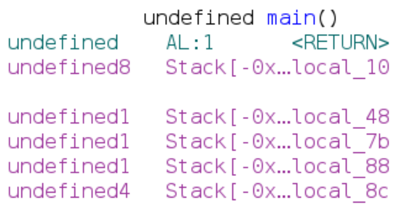
Listing 2. Source calculation test

5.1. Source size calculation

Our 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 works for initialized variables and cases where there is a reference to each of the variables. It 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.

An example of direct referencing into a variable is seen in Listing 2 and Figure 6.

Listing 2 shows a direct reference in the middle of a character array via static assignment within a loop. Figure 6 shows the stack layout 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 source code, further investigation is needed to detect these cases.



```
                 undefined main()
undefined        AL:1          <RETURN>
undefined8       Stack[-0x…local_10

undefined1       Stack[-0x…local_48
undefined1       Stack[-0x…local_7b
undefined1       Stack[-0x…local_88
undefined4       Stack[-0x…local_8c
```

Figure 6. Disassembly of the stack layer for the source test

5.2. Uninitialized structs

When structs are uninitialized, space on the stack frame is allocated for the whole struct, so individual data structures in a struct cannot be easily differentiated. To do so, a reference to each data structure's location in the code is needed. Absent these references, calculating the distance between variables could result in calculating space meant for another data structure within the struct.

There is no good solution. One could search for clues about the struct. Information from functions such as *sizeof()* can provide clues to the length of a data type. Given such functions, the compiler may statically assign the data type's size as an integer in the binary. These statically assigned values provide a direct means of determining the space allocated to a variable.

5.3. Instruction set coverage

ODSS 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 offsets remains untested. There are many other instructions that were not handled; this is an area for future work.

5.4. Conclusion

We demonstrate ODSS, automated support for static analysis of binary programs to search for buffer overflow vulnerabilities. The bottom-up tracking process showed how to use the sinks and sources found in binary code to discover potential vulnerabilities. ODSS was successful against test cases in the Juliet test suite and identified buffer overflow vulnerabilities in ten *libc* functions. The selected Juliet test cases allowed us to test variations of a sink's usage; however, these tests are short code sequences intended to illustrate vulnerabilities and do not reflect many challenges associated with larger real-world applications. Further research is required to enhance the capabilities of ODSS. Future work involves use of a more realistic test suite and open source programs with known buffer overflows. This will require further refinement of ODSS for use on a wider range of systems.

The ODSS code is available at: https://github.com/ecw0002/Ghidra-based-Static-Analysis-Tool-for-Detecting-Stack-Based-Buffer-Overflows/

Disclaimer Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the Department of the Navy or the U.S. Government.

References

[1] J. P. Anderson, "Computer security technology planning study," Tech. Rep. ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, October 1972.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. IEEE Symp. on Security and Privacy*, pp. 48–62, May 2013.

[3] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symp. on Security and Privacy*, pp. 138–157, May 2016.

[4] MITRE, "Common vulnerabilities and exposures." https://cve.mitre.org/, May 2022.

[5] E. C. Wikman, "Static Analysis Tools for Detecting Stack-based Buffer Overflows," Master's thesis, Naval Postgraduate School, Monterey, California, June 2020.

[6] National Security Agency, "Ghidra API." http://ghidra.re/ghidra_docs/api/, 2020.

[7] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, vol. 45, pp. 88–90, Oct 2012.

[8] National Security Agency, "Ghidra." https://www.nsa.gov/resources/everyone/ghidra/, 2019.

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

[10] C. Eagle and K. Nance, *The Ghidra Book*. San Francisco, CA: No Starch Press, September 2020.

[11] P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*. US Department of Commerce, NIST, 2018.

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

[13] 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.

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

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

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

[17] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.

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

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

[20] 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 Ann. Computer Software and Applns. Conf.*, vol. 2, pp. 450–459, July 2015.

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

[22] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. 7, pp. 65889–65912, 2019.

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

[24] die.net, "Linux programmer's manual." https://linux.die.net/man/3/, June 2022.

[25] AlephOne, "Smashing the stack for fun and profit," *Phrack*, vol. 7, August 1996, http://www.phrack.org/issues.html?issue=49.

[26] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. A.C.M.*, vol. 15, no. 12, pp. 1053–1058, 1972.

[27] C. Eagle, *The IDA Pro Book*. No Starch Press, 2008.

[28] Cerbero Labs, "Cerbero suite: The hacker's multitool." https://cerbero.io, August 2022.

[29] rohitab.com, "API Monitor." http://www.rohitab.com/apimonitor, 2012.

[30] F. E. Allen, "Control flow analysis," *SIGPLAN Notices*, vol. 5, pp. 1–19, July 1970.

[31] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, Nov. 2009.

[32] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," *NIST SP 800-142*, 2010.