A Hybrid Method of Defense against Buffer Overflow Attacks

Annapurna Dasari & Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ 85287
Email: dannapurna@yahoo.com, partha@asu.edu

Abstract

Buffer Overflow Attacks that exploit memory overruns in a variety of ways have been the most effective and difficult to prevent, methods of compromising system security. The root cause for exploitation of these vulnerabilities is the lack of availability of allocated size information of buffers at runtime. Consequently no bound checking is done in standard C library functions that are the most common interfaces for buffer manipulation leaving most C programs vulnerable to buffer overflow attacks. Ensuring proper bound checking at these interfaces can help prevent buffer overflow attacks.

Pure static approach to bounds checking does not work, as it is not possible to know the allocated size of dynamically allocated buffers at compile time. On the other hand, relying on pure dynamic approaches for collecting allocated size information incurs high runtime overhead. A hybrid approach that collects buffer bound information using static and dynamic methods and ensures enforcement of these boundaries at runtime can be very effective in preventing buffer overflows. Such a hybrid approach lowers the overhead of obtaining bound information while improving the accuracy of the information obtained. Tests on the implemented hybrid defense method promise efficient prevention and complete coverage of various buffer overflow attacks.

Keywords: Information Security, Software Security.

1. Introduction

Buffer Overflow vulnerabilities are being persistently exploited to successfully penetrate into system security. About 50% of the attacks reported by CERT [1] are based on buffer overflow vulnerabilities. They dominate the class of remote penetration attacks, where-in an attacker exploits a buffer overflow vulnerability by feeding a well crafted oversized input to the vulnerable program, thereby injecting and executing the code of his choice.

A buffer is a contiguous block of computer memory that holds multiple instances of the same data type. In C language, the word buffer commonly refers to character arrays (static) and character pointers (dynamic). A Buffer Overflow occurs when a buffer is written beyond its maximum allocated size, thus causing the memory immediately following the end of the buffer to be overwritten. The overflow can inject foreign (attack) code into an unsuspecting process and then hijacks control of that process to execute the injected code. The hijacking of control is usually accomplished by overwriting code pointers (like return addresses on the process stack, function pointers, parameters to relevant system calls or library calls) in the process memory [2].

Programs written in C language have always been plagued with Buffer Overflow vulnerabilities. This is because C language does not automatically check for array out-of-bounds condition and illegal pointer references. Second important reason being, most of the common library functions in C - string manipulation and input functions (scanf and printf and gets family of functions) are exploitable, i.e., they do not check for overflows while writing to buffers. Buffer Overflows under normal conditions cause the program to crash, however an attacker can exploit these vulnerabilities to accomplish malicious tasks. Therefore it is left to the programmer to explicitly check for such attacks, which is difficult and sometimes impossible (for example, gets() may encounter a large string and assign past the input buffer limitations.)

The methods of defense against Buffer Overflow attacks can be broadly classified as static and dynamic. Static methods of defense use source code analysis to detect potential buffer overflow vulnerabilities, which can possibly be exploited at runtime to launch an attack. Dynamic methods of defense prevent stack based buffer overflow attacks by checking if the known attack targets (like function return address or old base pointer on the stack) have been tampered with, before returning control to the target.

A Program can be completely defended against Buffer Overflow attacks when it can be ensured that every buffer manipulation function respects the maximum allocated size bound of the buffer. This leaves us with two problems:

1. To obtain the maximum allocated information of a buffer
2. To make this information available to all buffer manipulation functions and ensure that they respect the boundaries of the buffer.

The solution presented in this paper addresses the above problems using a hybrid approach. It uses static analysis to obtain the allocated size of each buffer in the program and the dynamic counterpart (that consists of a library of safe wrappers to unsafe buffer manipulation functions) makes use of this information and ensures that every buffer manipulation function respects the maximum allocated size bound of a buffer.

2. Motivation

In spite of the numerous methods of defense against buffer overflow attacks, about 50% of the remote attacks are based on buffer overflow vulnerabilities. This implies that the defense methods are not complete and that they can handle only a known subset of buffer overflow attacks. Buffer Overflows can be completely prevented if it can be ensured at runtime that no buffer is written beyond its maximum allocated size. Ensuring this, using pure static approach is not possible because of two reasons – firstly the static analyzer does not have allocated size information of all buffers and also the static analyzer relies on the programmer to prevent buffer overflows at runtime.

Most of the dynamic approaches proposed so far rely on post overflow related conditions to detect and prevent buffer overflow attacks. Such a protection mechanism relies on known attack techniques and targets and hence can be subverted by new unknown attacks. The crux of the problem is to prevent overflow from occurring itself rather than trying to stop the attack upon detecting an overflow. The hybrid approach proposed in this paper prevents buffer overflows by not allowing any buffer to be written beyond its maximum allocated size.

3. Related Work

3.1 StackGuard

The StackGuard compiler [3] is the most well known dynamic method of defense against buffer overflow attacks. It is designed to detect and stop stack based buffer overflow attacks targeting the return address on the stack. It guards the return address by placing a dummy value (canary value) between the return address and the stack data just before transferring control to a function. StackGuard protection can be subverted if the attacker can guess the dummy value, or by abusing a pointer to the return address [4]. It does not address other types of overflows on stack and elsewhere.

3.2 StackShield

This is a compiler patch for GCC [5], which is also based on the idea of protecting the return address on the stack. It implements three types of protection; two of them defend against overwriting of the return address and one against overwriting of function pointers. It basically implements all of them using auxiliary stacks or global variables to maintain copies of the original contents i.e. contents before function calls and then compares the respective contents before returning control, to determine if the return address or function pointers have been tampered with. Denial of service, memory protection of auxillary stacks and global variables, recompilation of code and limited nesting depth are the shortcomings of this approach [4][6].

3.3 Propolice

Propolice is a GCC patch [7] that is perhaps the most sophisticated compiler based protection mechanism. It borrows the idea of protecting the return address with canary values from StackGuard. Additionally it protects stack allocated variables by rearranging the local variables so that character buffers are always allocated at the bottom, next to the old base pointer, where they cannot be over flown to harm any other local variables. Denial of service and recompilation of code are the shortcomings of this approach. Moreover its protection is also limited to known stack based buffer overflow attacks only.

3.4 Libsafe/Libverify

This tool [8] is similar to the solution proposed in this paper as it also provides a combination of static and dynamic protection. Statically it patches exploitable buffer manipulations functions in standard C library. A range check is done by a safe wrapper function before proceeding with the actual operation, which ensures that the return address and the base pointer cannot be overwritten. Further protection is provided with Libverify using a dynamic approach similar to Stack Guard. It differs from the proposed solution in the way it determines the bound for a character buffer, it basically uses the old base pointer address on the stack as the boundary value and ensures that all buffer manipulation operations respect this boundary. This upper bound based protection still allows function pointers on the stack to be overwritten. Its protection is limited to stack based overflow attacks targeting the return address.

3.5 TIED, LibsafePlus

This is a newly developed tool for runtime buffer overflow protection. The idea of their protection method is similar to that presented in this paper; that is they first collect the size information of buffers in the program and then use it to detect overflows via function call interception as in Libsafe. They use a tool called TIED: Type Information Extractor and Depositor [9] that uses compiler generated debug information to obtain size information of static buffers (arrays). They address the problem of obtaining size information of dynamically allocated pointers by intercepting dynamic memory allocation functions like malloc. This method fails to obtain size of pointers assigned using pointer arithmetic expressions. Since pointer arithmetic is a more serious and common challenge for determining buffer size, this method may not be very effective in defending against buffer overflow attacks specifically against those that overflow vulnerable pointers.

3.6 LClint

This is a static analysis tool [10], which uses source code annotations to detect potential buffer overflow vulnerabilities. It uses the information provided in semantic comments to perform lightweight and efficient static analysis. While having a low false positive rate, this tool suffers from low true positives (that is it is not able to detect all likely buffer overflow vulnerabilities) [5]. It requires further enhancement as far as the security part is concerned. Moreover it suffers from the inherent limitation of static approaches of not being able to accurately predict the process's runtime state.

3.7 Return Address Defender

This tool [11] also intends to protect against return address based stack overflow attacks by copying the return address as in stack shield, but ensures protection of its copy of the return address. It can protect against long jump

pointer based attacks too, but incurs a greater overhead. This tool is very efficient in detecting return address based attacks however it cannot defend against any other kind of buffer overflow attack.

4. Design and Approach

The Hybrid Approach proposed in this paper uses static analysis and dynamic enforcement to ensure that no character buffer is written to, beyond its maximum allocated size. In order to do this, one needs to obtain the allocated size information of each buffer in the program and make this information available at runtime. This is done via static analysis of the source code. The static analyzer proposed in this paper obtains size information of statically allocated buffers and facilitates for obtaining size information of dynamically allocated buffers by inserting statements into the source code. These statements calculate allocated size at runtime. It also facilitates the availability of the size information collected, at runtime, by inserting appropriate metadata access operations in the source code. The dynamic counterpart of the solution ensures enforcement of buffer boundaries at runtime. The dynamic counterpart consists of a library of safe wrapper functions to unsafe buffer manipulation functions of standard C library. The safe counterparts perform bound checking using the metadata (allocated size of buffer) made available by static analyzer to prevent overflows during buffer write operations. Since most buffer manipulation operations are done via standard C library buffer manipulation functions, we believe that ensuring proper bound checking at these interfaces will help prevent buffer overflows.

4.1 Extracting Allocated Size Information

Static analysis of source code is used to obtain size information of buffers defined in the program. Static analysis of source code provides a lightweight method to obtain and access buffer size information at runtime Allocated sizes of static buffers are obtained by simply scanning the input source code. While the process of obtaining size information of dynamically allocated buffers (pointers) is more involved and is dependent on the method of allocation. A pointer can be allocated in 3 different methods: (1) Using dynamic memory allocation functions like malloc, calloc of standard C library. (2) Using pointer arithmetic expressions consisting of pointers and numeric constants. (3) Arbitrary assignment to an arbitrary pointer. Size information of pointers in the first case is obtained at runtime using metadata enabled wrapper functions to malloc family of functions. Calls to malloc family of functions in the source code are transformed to equivalent calls to the corresponding metadata enabled wrappers. The wrapper functions call the actual standard C memory allocation function and post the allocated size to the metadata table, thus making the size information available at runtime. For dynamic allocations using pointer arithmetic expressions, the static analyzer attempts to evaluate the resulting size as much as possible statically. It then inserts a statement(s) into the source code that take care of posting the resulting size to the metadata table at runtime. The process of extracting buffer size information understood by an example presented in Figure 1below.

Figure 1: Example 1

Input Source Code:

```
main() {
    char buff[100];
    char *ptr,
    ptr1 = malloc(200);
    ptr = ptr1;
    ptr = buff +6;
  }
```

Modified Code Metadata Enabled Code:

Note: Statements inserted/modified by the static analyzer are shown in italics

```
// include the header file for solutions safe library functions  and utility functions
#include <solution.h>
main() {
// metadata key value table declaration
    keyValueTable  pKVT[3];
//populate table with values from INI file
    populate_metadata(pKVT, 3);
    char buff[100];
    char *ptr;
// change malloc call  to equivalent safe_malloc call
   *ptr1 = safe_malloc(pKVT, 3, 3, 200);
    ptr = ptr1;
// first get size of  buffer ptr1
    int dummyVarMain =getbuffersize( pKVT, 3, 3);
 // Then post this size to the metadata table
    postbuffersize(pKVT, 3, 2, dummyVarMain);
    ptr = buff +6;
// size of buff is known at compile time, the size is  //evaluated completely and posted to the metadata //table
    postbuffersize(pKVT, 3, 2, 94);
}
```

Output Metadata INI file:

```
# section name corresponds to function name
[main]
buff = 100
# -1 size indicates pointers.
ptr = -1
```

It is difficult to accurately determine the allocated size of the pointer in arbitrary assignments. The allocated size in this case is determined by the available heap size at the time of allocation. A pointer assigned in this way can rarely be exploited to launch a targeted buffer overflow attack because the chances of guessing the precise position and the allocated size is minimal even after a number of unsuccessful attempts. However it is possible to launch denial of service attacks using arbitrary pointer overflow causing the process heap to get corrupted. Solution offers protection against such attacks by using the available heap

size as the upper bound, while determining the safety of operations writing to arbitrary pointers in the program.

4.2 Obtaining Buffer Size at Runtime

The solution provides utility functions: *getbuffersize()* and *postbuffersize()* that get and set buffer sizes at runtime. They perform simple lookup operations on the metadata key-value table to get/set size of the subject buffer. While evaluating pointer arithmetic expressions, the static analyzer inserts calls to *getbuffersize* and postbuffersize functions in the source code for evaluating and posting the resulting size, as illustrated in the previous example. The safe buffer manipulation functions call the *getbuffersize* function to obtain allocated size of the subject buffer, while the safe buffer allocation functions call the *postbuffersize* function to post the allocated size to the metadata table. The common metadata parameters required by these functions are: pointer to the metadata key-value table, size of the table and index of the subject buffer in the table. The buffers are indexed according to their order of declaration. All the metadata parameters required by these functions are known at compile time and the static analyzer takes care of inserting them in the source code wherever required.

4.3 Safe Library

Our solution provides a library of safe functions that ensure proper bound checking at buffer manipulation interfaces during runtime. The safe library provided by the proposed solution consists of safe buffer manipulation functions and metadata enabled buffer allocation functions. Any function that writes to a buffer without proper bound checking in considered an unsafe buffer manipulation function The safe buffer manipulation functions are wrapper functions to their unsafe counterparts, which perform proper bound checking. The safe functions retrieve buffer bound information (allocated size) from the metadata made available by the static analyzer to perform bound checking. A safe function detects a buffer overflow attempt by comparing the size of the data to be written to the buffer with the buffer's allocated size.

Upon detecting an overflow attempt it will either continue the write operation with safe arguments or will abort the operation and terminate program execution. If the safe function is configured to not to terminate the program execution upon detecting an overflow attempt, it will ensure that the overflow attempt will not succeed by truncating the data to be written to the buffer to that of the buffer's allocated size. The intended unsafe buffer write function is then called with the newly created truncated safe parameters.

The behavior of the safe function upon detecting an overflow attempt is a configurable option. The solution presented in this paper, considers the data used to overflow a buffer as non malicious as long as the buffer boundaries are respected. Hence by default it does not abort execution of the program or the write operation upon detecting an overflow attempt. This ensures protection from denial of service attacks launched via buffer overflows. However the user can configure it to abort program execution if he thinks that the overflow data is malicious and should never be copied.

The safe library is similar in concept to that of interception, but intercepted functions do not serve our purpose, as we need extra metadata parameters to obtain the buffer size. Also any program that calls our safe functions must go through static analysis; this is another important reason for introducing a new library of safe functions instead of using library call interception in this solution.

The static analyzer takes care of modifying calls to unsafe buffer manipulation functions to equivalent calls to their safe counterparts. It uses a configuration file that contains list of unsafe functions, their corresponding safe counterpart, and the function call transformation details. Based on the entries in the configuration file, it inserts the required metadata parameters and modifies the function name. The example shown in figure 2 illustrates the function call transformation done by the static analyzer.

The solution provides safe wrappers to certain most commonly used unsafe buffer manipulation functions of standard C library. However end users can extend the protection offered, by writing such a safe wrapper to any user defined buffer manipulation function in their application that is likely to be exploited to launch a buffer overflow attack. The static analyzer will automatically take care of transforming the calls to the unsafe user defined buffer manipulation function to that of its safe wrapper.

Figure 2: Example 2

Input Source Code:

```
int main() {
char buff[100];
char *p, *p1;
gets(buff);  }
```

Modified Code Metadata Enabled Code:

Note: Statements inserted/modified by the static analyzer are shown in italics

```
// include the header file for solutions safe library
// functions  and utility functions
#include <solution.h>
main() {
keyValueTable pKVT[3];
populate_keyvaluetable(pKVT, 3);
char buff[100];
char *p, *p1;
// Replaced safe function call
safe_gets(pKVT, 3, 1 , buff);
}
```

The buffer allocation functions of standard C library need to be replaced by metadata enabled counterparts of the solution's safe library to ensure the availability of allocated size information. The metadata enabled buffer allocation functions are also wrapper functions that post the allocated size to the metadata table. They can be considered as runtime counterparts to the static analyzer which provides the size information of static buffers.

5. Results

5.1 Test Scenarios and Setup

The solution has been tested in different scenarios. These tests assess the functionality and performance of the solution in different attack and worst-case scenarios.
All the tests were performed on a Linux machine that runs Suse Linux kernel version 2.4.21 on a 2GHz Intel Pentium laptop. Gcc version used is 3.3.1

5.2 Attack Coverage Test

In this setup, we test the ability of the proposed solution to prevent buffer overflow attacks in different attack scenarios. These scenarios cover most of the possible methods of launching a buffer overflow attack. The attack scenarios are chosen from publications about buffer overflow attacks, more specifically from John Wilander's Masters Thesis [6]. The attack techniques chosen are generic and are not targeted towards exploiting vulnerable features of a specific solution.
The tests are conducted using exploits that use the techniques mentioned below. The attacks can be classified into two categories based on the method of overflow.

5.2.1 Attack Targets:

The following common attack targets have been chosen for launching buffer overflows in this test.

1. **Target T1:** Function return address stored on the stack.
2. **Target T2:** Base Pointer on the stack.
3. **Target T3:** Program defined function pointer on the stack.
4. **Target T4:** Program defined function pointer on the heap or in BSS segment.
5. **Target T5:** Implicit function pointers are function pointers not declared in the program such as entries in the program's global offset table or address of the .dtors section. The global offset table has addresses of system functions that are called by the program; by overwriting these entries the attacker can execute code of his choice.
6. **Target T6:** Management Information header of a dynamically allocated chunk of memory. The management information header is overwritten to write attacker intended values to fd and bk fields of the header causing execution of malicious code.

5.2.2 Method 1:

Attack the target by overflowing a buffer, all the way to the target: technique is to attack the target directly by overflowing a vulnerable buffer to overwrite all the memory between the vulnerable buffer and the intended target address.

5.2.3 Method 2:

Attack the target by overflowing a buffer, to redirect an adjacent pointer to the target: technique is to attack the target by overflowing a vulnerable buffer to overwrite an adjacent pointer causing it to point to the intended target address. The attacker then uses the redirected pointer to alter the target accordingly.

Table 1 shows the results of this test – performance of each defense method with exploits targeting the targets mentioned above using method1 and method2. The results presented are based on the evaluation of the theoretical concepts behind each solution and on their performance, when tested against different exploits.

The numbers in the table reveal that the solution is the most effective defense method with 100% attack coverage. This proves our claim that the proposed defense is not attack specific and that it has the ability to catch any kind of buffer overflow attack. Propolic and stack shield follow next with about 50% coverage; the poor performance of the rest of the defense methods highlight their attack specific nature.

Table 1: Results of Test 1

Defense Method	T1	T2	T3	T4	T5	T6	Attacks Detected or Prevented
Proposed Solution	PP	PP	PP	PP	PP	PP	12 (100%)
Libsafe / LibVerify	HH	HH					4 (33 %)
Stack Guard Random XOR Canary	HH	H					3 (25%)
Stack Guard Terminator Canary	H						1 (8%)
Stack Shield Global Ret Stack	PP		HH	HH			6 (50%)
Stack Shield Range Ret Stack	HH		HH	HH			6 (50%)
RAD	HH						2 (17%)
Propolice	HP	HP	PP				6 (50%)

Note: Prevent (P) implies that the intended attack was prevented without terminating the program. Halt (H) implies that the intended attack was stopped by terminating the program. Blank column indicates that the attack was neither detected nor prevented by the defense method. Two entries in each of the table columns indicate the action taken by the defense method in protecting the corresponding target when attacked using method1 and method 2 resp.

5.3 Micro Bench Test

This test assesses the micro bench performance of the solution. The overhead imposed by the solution for initializing and maintaining the metadata is measured. Also the overhead imposed by seven of the solution's safe library functions is measured and compared to that of Libsafe counterparts. Timing measurements are done using wall clock elapsed time as reported by gettimeofday.

Table 4: Initialization Overhead vs. Number of Buffers in the Test Program

Number of Buffers	Initialization Overhead
100	250us
200	256us
500	263us
750	265us
1000	270 us

The initialization overhead is measured as the time taken by to populate the metadata table in the program with key-value entries from its respective metadata INI file. It can be observed from Table2 that the initialization overhead remains more or less constant as the numbers of buffers in the program increase. This is because most of the processing time is taken for signature verification and file open operations in the populate_metadata function. However this overhead is optional, the end user can chose to allow the static analyzer to populate the metadata table at compile time itself by inserting a set of assignment statements. When this option is enabled the initialization overhead was reduced to 10 microseconds for a program with 100 buffers. We observed that the initialization overhead in either case is much less than that of Propolice, Libsafe or any instrumentation based defense mechanism
The metadata maintenance overhead is assessed by measuring the time taken by postbuffersize method. The postbuffersize method imposes a negligible constant overhead of about 0.1 microsecond per call. The number of calls to the postbuffersize method is estimated to be around 1000 in the worst case.

The processing times of seven functions of the solution's safe library were measured. We used standard 256 byte arguments to the functions while measuring their processing times. The processing times of the original unsafe counterparts and Libsafe counterparts of each of these seven functions were also measured using the same arguments and under similar conditions.

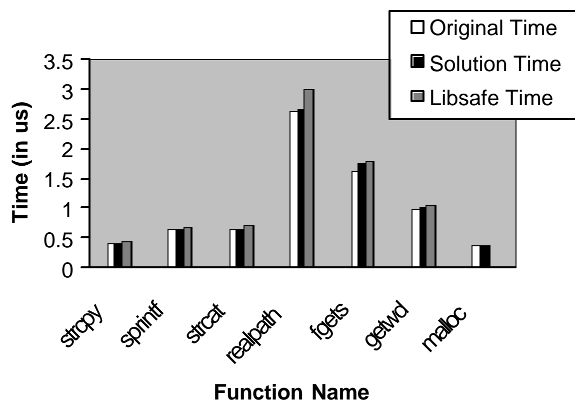

Figure 3: Results of Micro Bench Test

We can observe that the difference between the processing times of the solution's safe library functions and that of the original functions is negligible in spite of the additional bound checking. In some cases the solution's functions as in sprintf outperform the original versions. This can be attributed to the low-level optimizations done in solution's safe library functions. They perform optimizations similar to that of the Libsafe functions.

However we can observe a marked difference between the processing times of the solution's safe functions and that of the Libsafe functions, this can be attributed to the greater bound checking overhead of Libsafe functions. This is because Libsafe functions need to check for applicability before obtaining bounds information, also the process for obtaining bounds is more involved than just reading a particular metadata table entry as in the solution's safe library.

The figure illustrates that the solution's safe malloc function incurs a slight overhead of 0.02 micro seconds which can be attributed to the overhead of inserting the allocated size information in the metadata table.

5.4 Macro Bench Test

This test measures the performance of the solution in applications that present worst-case macro bench scenarios. Based on the way the solution was designed, different worst-case situations were identified and incorporated into three programs against which the solution was tested. Figure 3 shows the execution time of the programs when executed (i) without any security measure (ii) using Libsafe (iii) using solution. The execution times reported are the real time execution times reported by the time function.

Program1 has 1000 character buffers, and performs 10,000,000 strcpy operations with arguments of size 256 bytes. All the buffers used in this program are statically allocated. In this program the metadata maintenance overhead is negligible, as the buffer sizes do not change at runtime. We can observe that the solution imposes an acceptable overhead of about 40%, which is equal to that of Libsafe given the worst-case situation involving 10,000,000 strcpy operations.

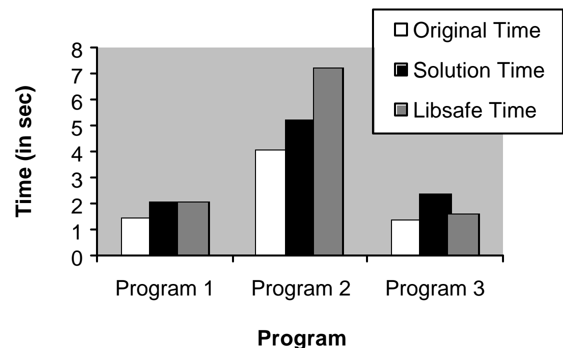

Figure 4: Results of Macro Bench Test

Program 2 performs a total of 50,000,000 strcpy operations on 1000 character pointers allocated dynamically using pointer assignment expressions. Program 2 presents more worse condition than Program1 for the solution as it involves additional 50,000,000 calls to postbuffersize function because of pointer manipulation before calling strcpy. However the results show a similar 30% overhead as in Program1, this can be attributed to the negligible overhead of the postbuffersize function and the low-level optimizations in solution's safe library functions. Contrary to our expectations the performance of Libsafe in this case

is much worse than the solution. This anomaly can probably be attributed to the bound determination process in Libsafe. Program 3 is used to assess the worst-case performance of the solution when using malloc for allocation. It consists of a 1,000 character buffers and makes 10,000,000 malloc calls in a loop. As the solution modifies each of these malloc calls to safe_malloc calls that write allocation information to the metadata table upon successful allocation. Results indicate a 40 - 45% overhead, which is acceptable given the worst-case situation.

The solution was tested against many other stand-alone application programs and the results indicate an acceptable performance overhead about 10% -15%. In most cases the performance of the solution was better than that of Libsafe that is so far considered the most efficient buffer overflow defense method.

6. Conclusions & Future Work

The solution presented in this paper provides an efficient method of defense against buffer overflow attacks. Its hybrid method builds on the strengths of static and dynamic methods providing a comprehensive defense against buffer overflow attacks. The solution was implemented and tested under various possible attack scenarios. The test results prove the efficiency of the solution in defending against different types of buffer overflow attacks.

The performance of the solution was also assessed using micro bench and macro bench tests. The comparison results prove the solution as the most efficient method of defense. Based on the solution's fundamental concept for defense and the results of test1, it can be confidently claimed that this solution can be very effective in defending against unknown buffer overflow attacks. We believe that the universal applicability (ability to defend against any kind of buffer overflow attack), minimal performance overhead, and above all the ability to defend against unknown buffer overflow attacks, can qualify it as a silver bullet defense method.

Present implementation of the solution can only protect stand-alone programs; the static analyzer needs to be extended to capture dependencies of a program to overcome this limitation. The static analyzer needs to be enhanced to be able to obtain bound information of multi level character buffers (pointer to a character pointer, pointer to pointer to character pointer). Future direction of work involves enhancement of the static analyzer to address the above limitations.

We also recognize that the performance of the solution can be greatly improved by customizing the static analyzer to the subject program. We are working on introducing customizability options which allow the user/author of the program to identify the subset of vulnerable buffers and functions in the program and limit the runtime protection only to this subset. Thus instead of protecting all buffers automatically (incurring high runtime overhead), it provides for intelligent protection by making use of the user/author's knowledge of the program.

7. References

[1] CERT/CC. Advisories 2002. http://www.cert.org/advisories.

[2] Aleph One. Smashing the stack for fun and profit. Phrack, 7(49), November 1996.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[4] Bulba and Kil3, Bypassig StackGuard and StackShield. Phrack (56), May 2000.

[5] D.Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, February 2000.

[6] J. Wilander and M. Kamkar, A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium,* pages 149-162, February, 2003.

[7] H. Etoh, GCC extension for protecting applications from stack smashing attacks. http://www.trl.ibm.com/projects/security/ssp.

[8] Arash Baratloo, Navjot Singh, Timothy Tsai. Transparent runtime defense against stack smashing attacks. In Proceedings *of the USENIX Annual Technical Confererence*, pages 251 -262, June 2000.

[9] Kumar Avijit, Prateek Gupta and Deepak Gupta, TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *proceedings of the 13th USENIX Security Symposium*, pages 45-56, August 2004.

[10] David Larochelle and David Evans, Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of USENIX Security Symposium*, August, 2001.

[11] Tzi cker Chiueh and Fu-Hau Hsu, RAD: A Compile time solution to buffer overflow attacks. In *proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS),* April 2001.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*, pages 119–129, Hilton Head, January 2000.

[13] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. International Workshop on Automated and Algorithmic Debugging*, pages 13-26, 1997

[14] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proc. 11th Annual Network and Distributed System Security Symposium*, Feb 2004.