TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection

Kumar Avijit Prateek Gupta Deepak Gupta
Department of Computer Science and Engineering
Indian Institute Of Technology, Kanpur
Email: {avijitk,pgupta,deepak}@iitk.ac.in

Abstract

Buffer overflow exploits make use of the treatment of strings in C as character arrays rather than as first-class objects. Manipulation of arrays as pointers and primitive pointer arithmetic make it possible for a program to access memory locations which it is not supposed to access. There have been many efforts in the past to overcome this vulnerability by performing array bounds checking in C. Most of these solutions are either inadequate, inefficient or incompatible with legacy code. In this paper, we present an efficient and transparent runtime approach for protection against all known forms of buffer overflow attacks. Our solution consists of two tools: TIED (Type Information Extractor and Depositor) and LibsafePlus. TIED extracts size information of all global and automatic buffers defined in the program from the debugging information produced by the compiler and inserts it back in the program binary as a data structure available at runtime. LibsafePlus is a dynamic library which provides wrapper functions for unsafe C library functions such as `strcpy`*. These wrapper functions check the source and target buffer sizes using the information made available by TIED and perform the requested operation only when it is safe to do so. For dynamically allocated buffers, the sizes and starting addresses are recorded at runtime. With our simple design we are able to protect most applications with a performance overhead of less than 10%.*

1 Introduction

Buffer overflows constitute a major threat to the security of computer systems today. A buffer overflow exploit is both common and powerful, and is capable of rendering a computer system totally vulnerable to the attacker. As reported by CERT, 11 out of 20 most widely exploited attacks have been found to be buffer overflow attacks [1]. More than 50% of CERT advisories [2] for the year 2003 reported buffer overflow vulnerabilities. It is thus a major concern of the computing community to provide a practical and efficient solution to the problem of buffer overflows.

In a buffer overflow attack, the attacker's aim is to gain access to a system by changing the control flow of a program so that the program executes code that has been carefully crafted by the attacker. The code can be inserted in the address space of the program using any legitimate form of input. The attacker then corrupts a code pointer in the address space by overflowing a buffer and makes it point to the injected code. When the program later dereferences this code pointer, it jumps to the attacker's code. Such buffer overflows occur mainly due to the lack of bounds checking in C library functions and carelessness on the programmer's part. For example, the use of `strcpy()` in a program without ensuring that the destination buffer is at least as large as the source string is apparently a common practice among many C programmers.

Buffer overflow exploits come in various flavours. The simplest and also the most widely exploited form of attack changes the control flow of the program by overflowing some buffer on the stack so that the return address or the saved frame pointer is modified. This is commonly called the "stack smashing attack" [3]. Other more complex forms of attacks may not change the return address but attempt to change the program control flow by corrupting some other code pointers (such as function pointers, GOT entries, longjmp buffers, etc.) by overflowing a buffer that may be local, global or dynamically allocated. Many common forms of buffer overflow attacks are described in [4].

Due to the huge amount of legacy C code existing today, which lacks bounds checking, an efficient runtime solution is needed to protect the code from buffer overflows. Other solutions which have developed over the years such as manual/automatic auditing of the code, static analysis of programs, etc., are mostly incomplete as they do not prevent all attacks. A runtime solution is required because certain type of information is not available statically. For example, information about dynamically allocated buffers is available only at runtime. However, most current runtime solutions are unacceptable because they either do not protect against all forms of buffer overflow attacks, break existing code, or impose too high an overhead to be successfully used with common applications. An acceptable solution must tackle all of these problems.

In this paper, we present a simple yet robust solution to guard against all known forms of buffer overflow attacks. The solution is a transparent runtime approach to prevent such attacks and consists of two tools: TIED and LibsafePlus. LibsafePlus is a dynamically loadable library and is an extension to Libsafe [5]. LibsafePlus contains wrapper functions for unsafe C library functions such as `strcpy`. A wrapper function determines the source and target buffer sizes and performs the required operation only if it would not result in an overflow. To enable runtime size checking we need to have additional type information about all buffers in the program. This is done by compiling the target program with the `-g` debugging option. TIED (Type Information Extractor and Depositor) is a tool that extracts the debugging information from a program binary and then augments the binary with an additional data structure containing the size information for all buffers in the program. This information is utilized by LibsafePlus to range check buffers at runtime. For keeping track of the sizes of dynamically allocated buffers, LibsafePlus intercepts calls to the `malloc` family of functions. Our tools thus neither require access to the source code (if it was compiled with the `-g` option) nor any modifications to the compiler, and are completely compatible with legacy C code. The tools have been found to be effective against all forms of attacks and impose a low runtime performance overhead of less than 10% for most applications.

The rest of the paper is organized as follows. We present an overview of our approach in Section 2. Section 3 describes the implementation of TIED and LibsafePlus. This is followed by a description of performance experiments and results, in Section 4. Section 5 briefly discusses the related work done in the area of buffer overflow protection. Section 6 concludes the paper.

2 Basic approach

The steps in the protection of a program using TIED and LibsafePlus are shown in Figure 1. The key idea here is to augment the executable with information about the locations and sizes of character buffers. To this end, the program source must be compiled with the `-g` option which directs the compiler to dump debugging information regarding the sizes and types of all variables in the program in the generated executable binary. The next step is to rewrite the executable with the required information as an additional data structure in the form of a separate read-only section of the executable. This makes the information about buffer sizes available at runtime. The binary rewriting of the executable is done by TIED. LibsafePlus is implemented as a dynamically loadable library that must be preloaded for a process to be protected. To enable range checking, LibsafePlus provides wrapper functions for unsafe C library functions. Each such wrapper function checks the bounds of the destination buffer before performing the actual operation. For dynamically allocated buffers, LibsafePlus maintains an additional runtime data structure that stores information about the locations and sizes of all dynamically allocated buffers. In contrast to other approaches which are mainly compiler extensions, LibsafePlus does not require source code access if the program is compiled with the `-g` option and is not statically linked with the C library.

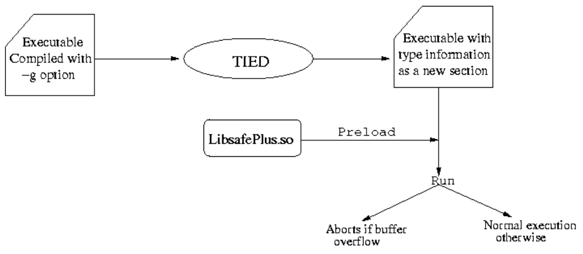
Executable Compiled with -g option
TIED
Executable with type information as a new section
LibsafePlus.so
Preload
Run
Aborts if buffer overflow
Normal execution otherwise

Figure 1: Rewriting of the binary executable by TIED and runtime range checking by LibsafePlus

LibsafePlus is implemented as an extension to Libsafe [5]. Libsafe is also a dynamically loadable library which provides wrapper functions for unsafe C library functions such as `strcpy()`. However, Libsafe protects only against stack smashing attacks. Even for stack variables, Libsafe assumes a safe upper bound on the size of a buffer instead of determining its exact size. Therefore, it is possible for the attacker to change variables in the program that are next to the buffer in memory. Unlike Libsafe, our tools offer full protection against all forms of attack and determine the exact sizes of *all* buffers. They have been tested extensively and have been found to be effective against all forms of buffer overruns. Our tools successfully prevented all the 20 different overflow attacks in the testbed developed by Wilander and Kamkar for testing tools for dynamic overflow attacks [6], while the original Libsafe could only detect only 6 of the 20 attacks.

In the following subsections, we describe in detail the design of Libsafe and our extensions to it. We first describe in Section 2.1, the protection mechanism used by Libsafe and then show in Section 2.2, how LibsafePlus extends the basic protection mechanism, to handle all forms of buffer overflow attacks.

2.1 Runtime range checking by Libsafe

The goal of Libsafe is to prevent corruption of the return addresses and saved frame pointers on the stack in the event of a stack buffer overflow. Libsafe does not guarantee protection against any other form of attack. To ensure that the frame pointers and the return addresses are never overwritten, Libsafe assumes a safe upper bound on the size of stack buffers, since it does not possess sufficient information to determine the exact sizes of stack buffers at runtime. The underlying principle is that a buffer cannot extend beyond the stack frame within which it is allocated. Thus the maximum size of a buffer is the difference between the starting address of the buffer and the frame pointer for the corresponding stack frame. To determine the frame corresponding to a stack buffer, the topmost stack frame pointer is retrieved and the frame pointers are traversed on the stack until the required frame is discovered.

Based on the above design, Libsafe is implemented as a dynamically loadable library which provides wrapper functions for unsafe C functions such as `strcpy()`. The purpose of a wrapper function is to determine the size of the destination buffer and check whether the destination buffer is at least as large as the source string. If the check fails, the program is terminated. Otherwise, the wrapper function simply calls the original C library function.

2.2 Extended runtime range checking by LibsafePlus

As seen above, Libsafe determines bounds on the size of stack buffers and prevents overwriting of frame pointers and return addresses. Although, it provides transparent runtime protection against buffer overflows it does so only for stack buffers. Also, for stack buffers the attacker is allowed to overwrite everything in the stack frame upto the frame pointer.

Our extension to Libsafe, LibsafePlus is able to thwart all forms of buffer overflow attacks. In order to perform precise range checking of global and local buffers, LibsafePlus uses the information about buffer sizes made available to it at runtime by TIED. If this information is not available, LibsafePlus falls back to the checks performed by Libsafe (no range checks for global buffers and upper bounds on sizes of local buffers). For range checking dynamically allocated buffers, LibsafePlus intercepts calls to the `malloc` family of functions and thus keeps track of the sizes of various dynamically allocated buffers.

3 Implementation

In this section, we describe the implementation of TIED and LibsafePlus. Sections 3.1 and 3.2 show how TIED extracts the type information from an executable and makes it available as a new section in the binary. Section 3.3 describes how LibsafePlus keeps track of the addresses and sizes of dynamically allocated buffers. Finally, in Section 3.4, we describe how LibsafePlus range checks buffers at runtime by intercepting unsafe C library functions.

3.1 Extracting type information

If the `-g` option is used to compile a program, the compiler adds type information about all variables to the executable in the form of special debugging sections. DWARF (Debugging With Arbitrary Record Format) [7] is the standard format for encoding the symbolic, source level debugging information. TIED uses the *libdwarf* consumer interface [8] to read the DWARF information present in the executable. For each function, information about all the local buffers is collected in the form of (offset from frame pointer, size) pair. In the current implementation, we extract information about character arrays only. For global buffers, the starting addresses and sizes are extracted. The members of arrays, structures and unions are also explored to detect any buffers that may lie within them. Figure 2 demonstrates a typical case of buffers within structures. TIED detects all the 40 buffers in this case.

```
struct s{
      char a[10];
      char b[5];
};
struct s foo[20];
```

Figure 2: Buffers within a structure

Buffers that appear inside a union may overlap with each other. For example, consider the variable `x` declared as in Figure 3. Here, the buffer `x.s2.b` partially overlaps with both `x.s1.a` and `x.s1.c`. The problem is to decide whether a string copy of 10 bytes at destination address `((void *)&x + 4)` should be permitted. If it is, it may be used by an attacker to overflow `x.s1.a` and write an arbitrary value to `x.s1.b`. On the other hand, if the string copy is not permitted, legitimate writes to `x.s2.b` may be denied. TIED, by default, takes the latter approach, in order to prevent all possible buffer overflows. However, it is possible to force TIED to take the former approach by specifying a command line option.

3.2 Binary rewriting

After extracting the type information from the DWARF tables in the executable, TIED first filters it to retain information only about variables that are character arrays. It then constructs data structures to store this information for efficient runtime lookup. These data structures are then dumped back into the executable file as a new read-only, loadable section. Currently TIED handles executable files in the ELF format only.

The type information available at runtime is organized in the form of several tables that are linked with each other through pointers, as shown in Figure 4. The top level structure is a type information header that contains pointers to, and sizes of a global variable table, and a function table. The global variable table contains the starting addresses and sizes of all global buffers. The function table contains an entry for

```
struct my_struct1{
char a[10];
void *b;
char c[10];
};
struct my_struct2{
void *a;
char b[16];
};
union my_union{
struct my_struct1 s1;
struct my_struct2 s2;
} x;
```

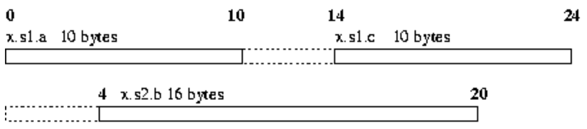

Figure 3: Overlapping buffers inside a union

each function that has one or more character buffers as local variables or arguments.[1] Each entry in the function table contains the starting and ending code addresses for the function, and the size of and a pointer to the local variable table for the function. The local variable table for a function contains sizes and offsets from the frame pointer for each local variable of the function or argument to the function that is a character array. The global variable table, the function table, and the local variable tables are all sorted on the addresses or offsets to facilitate fast lookup.

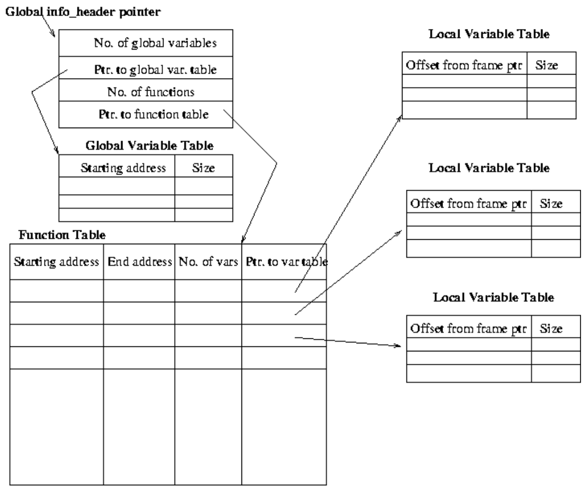

Figure 4: Data structures for storing type information

[1]An array can be an argument passed by value to a function if the array is part of a structure and the structure is passed by value.

After constructing these tables in its own address space, TIED finds a suitable virtual address in the target executable for dumping these data structures. The data structure is then "serialized" to a byte array, and the pointers are relocated according to the address at which the data structure will be placed in the target binary.

To ensure that addresses of existing code and data elements in the target binary do not change, the target binary is extended towards lower addresses by a size that is large enough to hold the type information data structures and is a multiple of the page size. The new data structure is dumped in this space. A pointer to the new section is made available as the value of a special symbol in the dynamic symbol table of the binary. Since this requires changes to the `.dynstr`, `.dynsym`, and `.hash` sections, and these sections cannot be enlarged without changing addresses of existing objects, TIED places the extended versions of these sections in the new space created, and changes their addresses in the existing `.dynamic` section. Figure 5 illustrates the changes made to the target binary.

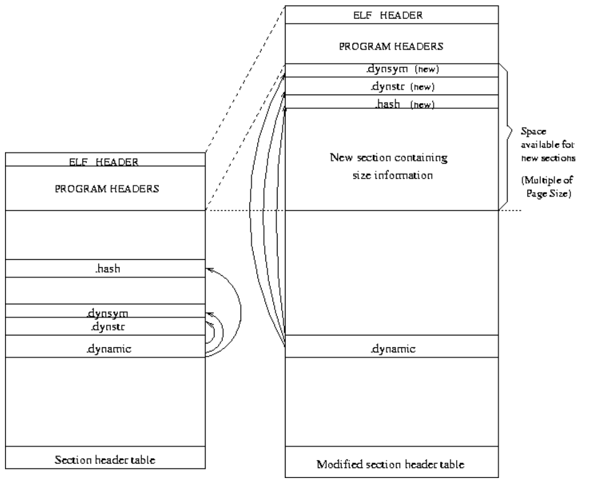

Figure 5: ELF executable before and after rewriting

3.3 Extracting size of heap buffers

By binary rewriting, all the buffers whose sizes are known at compile time can be protected from overflow. To capture the sizes of all dynamically allocated buffers, Libsafe-Plus intercepts all calls to the `malloc` family of functions, viz. `malloc`, `calloc`, `realloc` and `free`. In addition to calling the actual glibc function, the wrapper function records the starting address and the size of the chunk of memory allocated. The number of elements `nmem` in the buffer is also recorded. nmem is equal to 1 except for buffers allocated using calloc(nmemb, size), in which case it is equal to nmemb. LibsafePlus uses nmem to enforce a more rigorous size check.[2] For example, for the code below, an overflow will be detected if the tighter check is enforced.

```
char *buf = (char *)calloc( 5, 10 );
strcpy(buf, "A long string");
```

A red-black tree [9] is used to maintain the size information about dynamically allocated buffers. The tree contains a node for each buffer allocated using malloc, calloc or realloc. On freeing a memory area using free, the corresponding node is removed. Memory allocation for nodes in the red-black tree is done by a fast, custom memory allocator that directly uses mmap to allocate memory.

3.4 Intercepting unsafe functions and bounds verification

As outlined in Section 2, LibsafePlus works by intercepting unsafe C library functions. The wrapper functions attempt to determine the size of destination buffer. If the size of source buffer is less than that of the destination buffer, an actual C library function like memcpy or strncpy is used to perform the copying. An overflow is declared when the size of contents being copied is more than what the destination can hold, in which case the program is killed. If the size of the buffer can not be determined (for example, if TIED was not used to augment the binary and the buffer is either global or local), the default protection offered by Libsafe is provided.

To determine the size of the destination buffer, it is first checked whether the destination buffer is on the stack, simply by checking if its address is greater than the current stack pointer. If found on stack, the stack frame encapsulating the buffer is found by tracing the frame pointers. The function corresponding to the stack frame is searched in the function table present in the new section, using the return address from the stack frame above. Finally, the size of the buffer is found by searching in the local variable table corresponding to the function.

If the buffer is not on stack, it is checked whether it is on the heap by comparing its address with the minimum heap address. The minimum heap address is recorded by the malloc and calloc wrappers and is the address of the chunk allocated by the first call to malloc or calloc. The buffer is assumed to be on the heap if its address is greater than the minimum heap address. In this case, its size is determined by searching in the red-black tree.

Finally, if the buffer is neither on stack, nor on heap, it is searched for in the global variable table. If none of the above checks yields the size of buffer, the intended operation of the wrapper is performed. If the size of destination buffer is available, size of the contents of source buffer is determined. The contents are copied only if destination buffer is large enough to hold all the contents. The program is killed otherwise.

[2] A few programs have been found to fail when the rigorous check is applied. LibsafePlus, therefore, provides the strict check as an option that can be turned on using an environment variable.

4 Performance

We have tested LibsafePlus for its ability to detect buffer overflows as well as for the overhead incurred by loading LibsafePlus with applications. To test the protection ability of LibsafePlus, we used the test suite developed by Wilander and Kamkar [6]. This test suite implements 20 techniques to overflow a buffer located on stack, .data or .bss sections. The test suite executable was first modified using TIED. TIED detected all the global and local buffers declared in the test suite program. LibsafePlus was then preloaded while running the binary. All tests were successfully terminated by LibsafePlus when an overflow was attempted.

For testing performance overhead incurred due to LibsafePlus, we first measured overhead at a function call level. Next, the overall performance of 12 representative applications was measured. In the following subsections, we describe these tests and their results.

4.1 Micro benchmarks

In this section, we present a comparison of the execution times of various library functions like malloc(), memcpy() etc. for the following three cases.

- The test was run without any protection.
- The program was protected with Libsafe.
- The program was protected with LibsafePlus.

The tests were conducted on a 1.6 GHz Pentium 4 machine running Linux 2.4.18.

We present here the performance results for two most commonly used string handling functions: memcpy and strcpy. To measure the overhead of finding sizes of global and local buffers using the new section in the executable, we performed the following experiment. The test program contained 100 global buffers and 100 functions. Each function had 3 local buffers. The time required by a single memcpy() into global and local buffers was measured for varying number of bytes copied. As shown in Figure 6, we found a constant overhead of 0.8μs for memcpy() to global buffers. This translates to a 100% overhead for memcpy() upto 64 bytes and decreases to a 12% overhead for memcpy() involving about 1024 bytes. For local buffers, the overhead due to LibsafePlus is 2.2μs per call to memcpy() as shown in Figure 7. This includes the 0.9μs overhead due to Libsafe for locating the stack frame corresponding to the buffer.

To measure the overhead of finding size of a heap variable from the red-black tree, the test program first allocated 1000

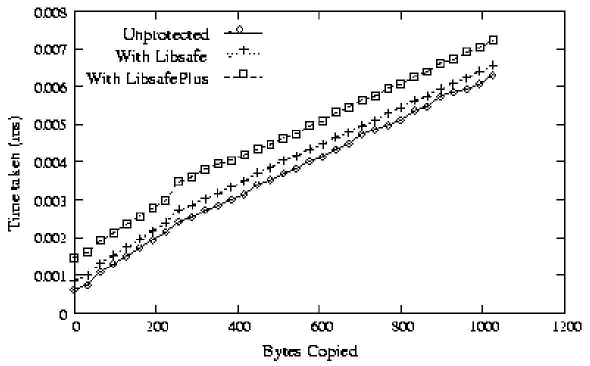

Figure 6: memcpy( ) to a global buffer

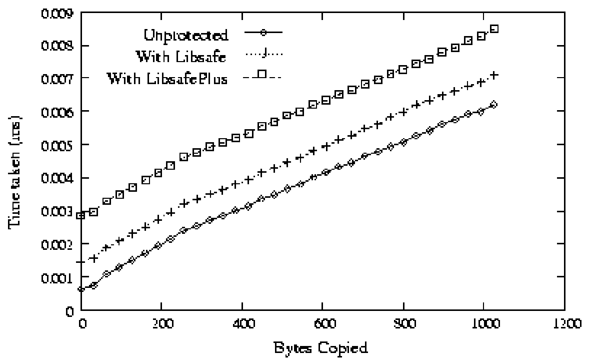

Figure 7: memcpy( ) to a local buffer

heap buffers. It then allocated another heap buffer and measured the time taken by one `memcpy()` to it. This represents the worst case performance as the buffer being copied to is the right most child in the red-black tree. As shown in Figure 8, the overhead due to LibsafePlus is 1.6μs per call to `memcpy()`.

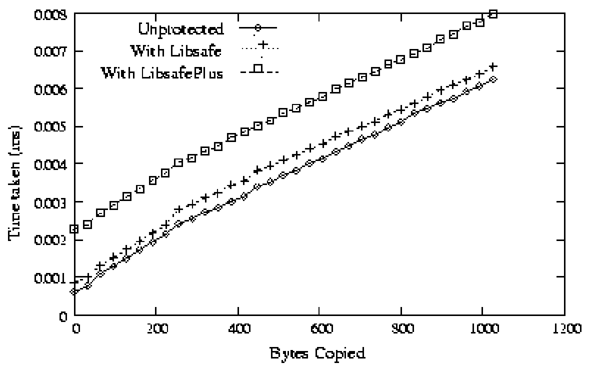

Figure 8: memcpy() to a heap buffer

We also measured the performance of LibsafePlus for calls to `strcpy()`. The testbed was similar to the one described earlier for `memcpy()`. Figure 9 shows the time taken by one `strcpy()` to a global buffer. The overhead drops from

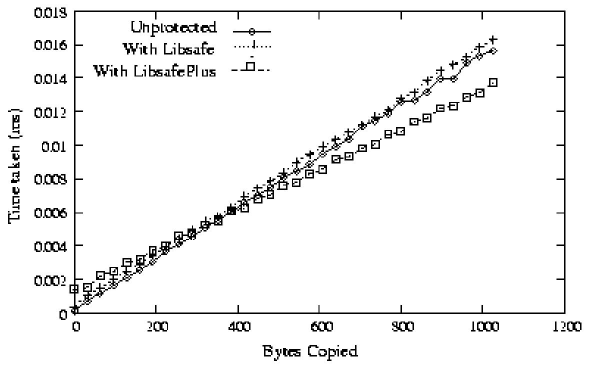

Figure 9: strcpy() to a global buffer

0.8μs for buffers of size 1 byte to 0 for buffers of about 400 bytes. This is because the wrapper function for `strcpy()` in LibsafePlus uses `memcpy()` for copying, which is 6 to 8 times faster than `strcpy()` for large buffer sizes. Figures 10 and 11 show similar results for `strcpy()` to local and heap buffers respectively.

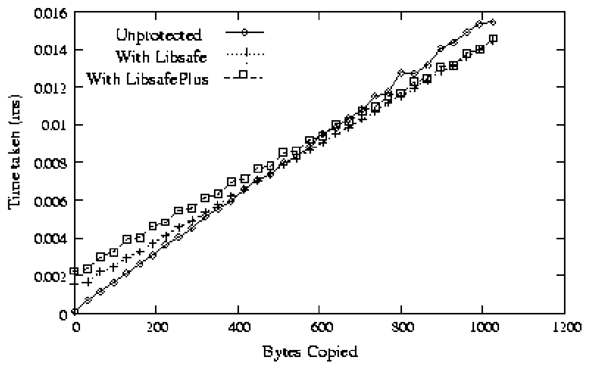

Figure 10: strcpy() to a local buffer

Next, we measured the overhead due to LibsafePlus in dynamic memory allocation. The insertion and deletion of nodes in the red-black tree is the primary constituent of this overhead. We measured the time required by a pair of `malloc()` and `free()` calls. The number of buffers already present in the red-black tree at the time of allocating the buffer was varied from 2^5 to 2^{21}. As shown in Figure 12, the time taken by LibsafePlus for `malloc()`, `free()` pair grows almost logarithmically with the number of buffers already present in the red-black tree. This is expected because of the $O(\log(N))$ time operations of insertion and deletion of nodes in a red-black tree.

4.2 Macro benchmarks

Next, we measured the performance overhead due to LibsafePlus using a number of applications that involve substantial dynamic memory allocation and operations like `strcpy()`

Application	What was measured
Apache-2.0.48	Connection rate, response time and error rate while requesting a large file from the web server.
Sendmail-8.12.10	Time to connect and connection rate achieved while sending a large message.
Bison-1.875	Time to parse a large grammar file and generate C code.
Enscript-1.6.1	Time to convert a large text file to postscript.
Hypermail-2.1.8	Time to process a large mailbox file.
OpenSSH-3.7.1	Time to transfer a large set of files using the loopback interface.
OpenSSL-0.9.7	Time to sign and verify using RSA.
Gnupg-1.2.3	Time to encrypt and decrypt a large file.
Grep-2.5	Time to perform a search for palindromes using back references on a large file.
Monkey	Connection rate, response time and error rate while requesting a large file from the web server.
Ccrypt	Time to decrypt a large file encrypted using ccrypt.
Tar	Time to compress and bundle a large set of files.

Table 1: Description of application benchmarks

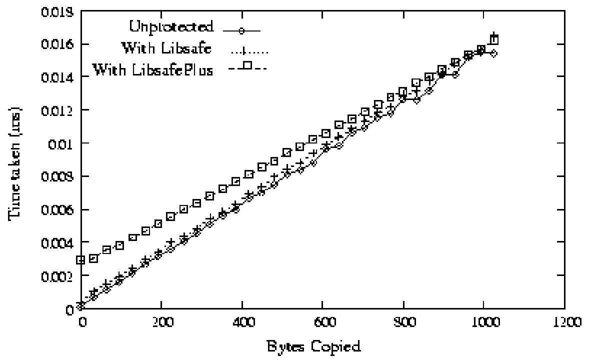

Figure 11: strcpy() to a heap buffer

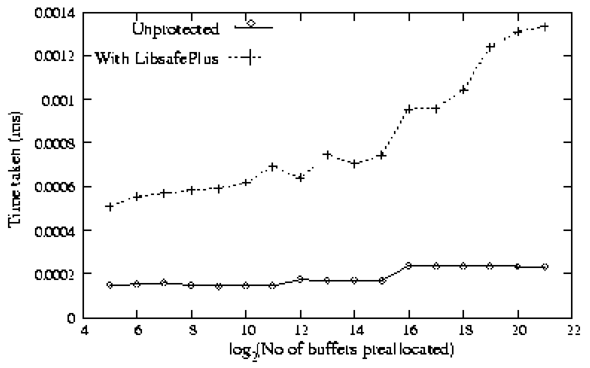

Figure 12: Performance overhead for malloc(), free() pair

to buffers. In all, a total of 12 applications were used to evaluate the overhead of LibsafePlus and Libsafe. Table 1 describes the performance metric used in each case. The performance overheads are shown in Figure 13. The graph shows normalized metric values with respect to the case when no library was preloaded. The overhead due to LibsafePlus was found to be less than 34% for all cases except for Bison. In 8 out of 12 applications, the overhead of LibsafePlus was within 5% of that of Libsafe. In case of Enscript, Grep and Bison, the slowdown observed is due to a huge number of dynamic memory allocations and string operations on heap buffers.

We now present a comparison of performance overhead of our tool with that of CRED [10] (strings only mode). As shown in Table 2, for 9 out of the 11 applications which have been used to measure the performance overhead of both the tools, LibsafePlus performs better than CRED. The slowdown observed for CRED, as compared to LibsafePlus, is significant for Apache, Enscript, Hypermail, Gnupg and Monkey.

Application	LibsafePlus	CRED
Apache	1.0X	1.6X
Bison	2.4X	1.2X
Enscript	1.3X	1.9X
Hypermail	1.1X	2.3X
OpenSSH	1.0X	1.0X
OpenSSL	1.0X	1.1X
Gnupg	1.0X	1.8X
Grep	1.3X	1.2X
Monkey	1.3X	1.8X
Tar	1.0X	1.0X
Ccrypt	1.0X	1.1X

Table 2: Performance overheads of LibsafePlus and CRED (strings only mode)

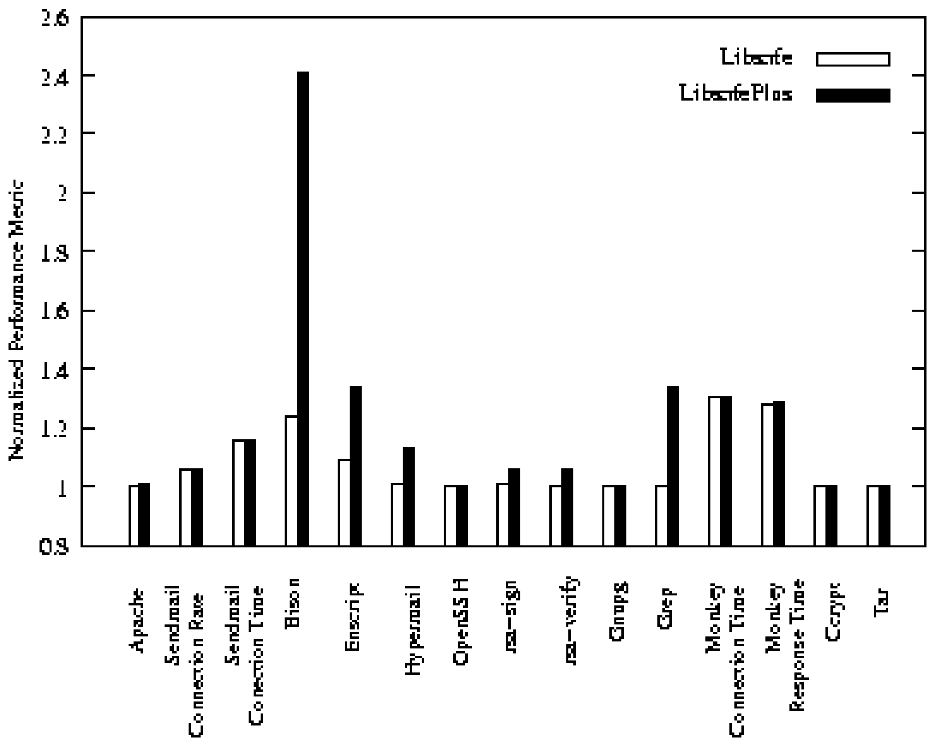

Figure 13: Macro performance overheads

5 Related work

In this section, we review the related work in the area of protection against buffer overflow attacks.

5.1 Kernel based techniques

The common feature used by the majority of buffer overflow attacks is the ability to execute code located on the stack. Solar Designer has developed a Linux patch that makes the stack non-executable [11], precisely to counteract the stack smashing attacks. The solution has some serious weaknesses. First, nested functions or trampoline functions, which are used by LISP interpreters, many Objective C compilers (including gcc), and most common implementations of signal handlers in Unix, require the stack to be executable. Second, the attacker does not require the code to be stored on a stack buffer for the exploit to work. Methods to bypass the non-executable stack defense have been explored by Wojtczuk [12].

PaX [13] is another kernel patch which aims to protect the heap as well as the stack. The idea behind PaX is to mark the data pages non-executable by overloading supervisor/user bit on pages and enabling the page fault handler to distinguish the page faults due to attempts to execute data pages. PaX also imposes a significant performance overhead due to additional work done by the page fault handler for each page fault. Although protecting the heap offers some additional protection but still it does not guarantee complete protection from all forms of attacks. For example, return-into-libc attacks are still possible.

5.2 Static analysis based techniques

Static analysis approaches to handling buffer overflows attempt to analyze the program source and determine if the program execution can result in a buffer overflow.

Wagner *et al.* formulated the detection of buffer overruns as an integer range analysis problem [14]. The approach models C strings as a pair of integer ranges (allocated size and length) and vulnerable C library functions are modeled in terms of their operations on the integer ranges. Thus, the problem reduces to an integer range tracking problem. The described tool checks, for each string buffer, whether its inferred length is at least as large as the allocated length. The tool is impractical to use since it produces a large number of false positives, due to lack of precision, as well as some false negatives.

The annotation based static code checker based on LCLint [15] by Larochelle and Evans [16] exploits the information provided in programs in the form of semantic comments. The approach extends the LCLint static checker by introducing new annotations which allow the declaration of a set of preconditions and postconditions for functions. The tool does not detect all buffer overflow vulnerabilities and often generates spurious warnings.

CSSV [17] is another tool for statically detecting string manipulation errors. The tool handles large programs by analyzing each procedure separately and requires *procedure contracts* to be defined by the programmer. A procedure contract defines a set of preconditions, postconditions and side-effects of the procedure. The tool is impractical to use for existing large programs since it requires the declaration of procedure contracts by the programmer. As for other static techniques, the tool can produce false alarms.

5.3 Runtime techniques

StackGuard [18] is an extension to the GNU C compiler that protects against stack smashing attacks. StackGuard enhances the code produced by the compiler so that it detects changes to the return address by placing a *canary* word on the stack above the return address and checking the value of the canary before the function returns. The *canary* is a sequence of bytes which could be fixed or random. The approach assumes that the return address is unaltered if and only if the canary word is unaltered. StackGuard imposes a significant runtime overhead and requires access to the source code. Techniques to bypass StackGuard protection are described by Richarte [19].

StackShield [20] is also implemented as a compiler extension that protects the return address. The basic idea here is to save return addresses in an alternate non-overflowable memory space. The resulting effect is that return addresses on the stack are not used, instead the saved return addresses are used to return from functions. As with StackGuard, the source code needs to be recompiled for protection. A detailed description of StackShield protection and techniques to bypass it were presented by Richarte [19].

Propolice [21] is another compiler extension which modifies the syntax tree or intermediate language code for the protected program. SSP (Propolice) aims to protect the saved frame pointer and the return address by placing a random canary on the stack above the saved frame pointer. In addition, SSP protects local variables and function arguments by creating a local copy of arguments and rearranging the local variables on the stack so that all local buffers are stored at a higher address than local variables and pointers. As for StackGuard and StackShield, it requires the recompilation of the source code. Although SSP protects against stack smashing attacks, it is vulnerable to other forms of attacks.

The memory access error detection technique by Austin *et al*. [22] extends the notion of pointers in C to hold additional attributes such as the location, size and scope of the pointer. This extended pointer representation is called the *safe pointer* representation. The additional attributes are used to perform range access checking when dereferencing a pointer or while doing pointer arithmetic. The approach fails to work with legacy C code as it changes the underlying pointer representation.

The backwards compatible bounds checking technique by Jones and Kelly [23] is a compiler extension that employs the notion of *referent objects*. The referent object for a pointer is the object to which it points. The approach works by maintaining a global table of all referent objects which maintains information about their size, location, etc. Furthermore, a separate data structure is maintained for heap buffers by modifying `malloc()` and `free()` functions. Range checking is done at the time of dereferencing a pointer or while performing pointer arithmetic. The technique breaks existing code and involves a high performance overhead for applications which are pointer and array intensive since every pointer or array access has to be checked at runtime.

The C Range Error Detector(CRED) [10] is an extension of Jones and Kelly's approach. CRED extends the idea of referent objects and allows the use of a previously stored out-of-bounds address to compute an in-bounds address. This is done by storing all the information about out-of-bounds addresses in an additional data structure on the heap. The approach fails if an out-of-bounds address is passed to an external library or if an out-of-bounds address is cast to an integer and subsequently cast back to a pointer. As for Jones and Kelly's technique, the tool involves a high performance overhead for pointer/array intensive programs since every access to a pointer has to be checked.

The type assisted dynamic array bounds checking technique by Lhee and Chapin [24] is also a compiler extension that works by augmenting the executable with additional information consisting of the address, size and type of local buffers, pointers passed as parameters to functions and static buffers. An additional data structure is maintained for heap buffers. Range checking is actually performed by modified C library functions which utilize this information to guarantee that overflows do not occur. As for other compiler based techniques, the solution is not portable and requires access to the source code of the program. It can be seen that our approach is very similar to Lhee and Chapin's approach. However, the main advantage of our approach is that it does not require compiler modifications and can work with the output of any compiler that can produce debugging information in the DWARF format.

PointGuard [25] is a pointer protection technique that encrypts pointers when they are stored in memory and decrypts them when they are loaded into CPU registers. PointGuard is implemented as a compiler extension that modifies the intermediate syntax tree to introduce code for encryption and decryption. Encryption provides for confidentiality only, hence PointGuard gives no integrity guarantees. Although, PointGuard imposes an almost zero performance overhead for most applications, it protects only code pointers (function pointers and longjmp buffers) and data pointers and offers no protection for other program objects. Also, protection of mixed-mode code using PointGuard requires programmer intervention.

One of the major drawbacks of all existing runtime techniques is that they require changes to the compiler. None of these techniques seem to have been adopted by any of the mainstream compilers so far. In contrast, our approach does not require any compiler modifications and can be used with any existing compiler. We feel that this may lead to widespread adoption of this technique in practice.

6 Conclusions and future work

In this paper, we have presented TIED and LibsafePlus. These are simple, robust and portable tools that can together guard against all known forms of buffer overflow attacks. Our approach is a transparent runtime solution to the problem of preventing buffer overflows that is completely compatible with existing code and does not require source code access. Experiments show that our approach imposes an acceptably low overhead due to the runtime checks in most cases.

There are certain cases which our approach is unable to handle. LibsafePlus can only guard against buffer overflows due to injudicious use of unsafe C library functions and not those due to other kinds of errors in the program itself. However, in most programs buffer overflows occur due to improper use of C library functions rather than erroneous pointer arithmetic done by the programmer. Moreover, guarding against erroneous pointer arithmetic implies protecting every pointer instruction which would incur a high performance overhead (as in CRED).

Also, LibsafePlus cannot handle dynamic memory allocated using `alloca`. `Alloca` is used to dynamically create space in the current stack frame, and the space is automatically freed when the function returns. The difficulty in handling `alloca` in LibsafePlus is that while memory allocation can be tracked by intercepting calls to `alloca`, it is not possible to track when a buffer is freed, since the freeing happens automatically when the function that called `alloca` returns. Variable sized automatic arrays (supported by gcc) present a similar problem.

Since LibsafePlus uses `mmap` for allocating nodes for the red-black tree, programs that use `mmap` for requesting memory at specified virtual addresses may not work with LibsafePlus.

A limitation of the current implementation is that size information for local and global buffers declared within dynamically loaded libraries is not available at runtime. We are currently extending LibsafePlus and TIED to address this issue.

TIED and LibsafePlus are available in the public domain and can be downloaded from http://www.security.iitk.ac.in/projects/Tied-Libsafeplus.

References

[1] CERT/CC. Vulnerability notes by metric. http://www.kb.cert.org/vuls/bymetric?open &start=1&count=20.

[2] CERT/CC. Cert advisories 2003. http://www.cert.org/advisories/#2003.

[3] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov 1996.

[4] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. DARPA Information Survivability Conference and Expo (DISCEX)*, Jan 2000.

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. USENIX Annual Technical Conference*, 2000.

[6] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proc. 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, Feb 2003.

[7] TIS Committee. DWARF debugging information format specification version 2.0, May 1995.

[8] UNIX International Programming Languages Special Interest Group. A consumer library interface to DWARF, Aug 2002.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2002.

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

[11] Solar Designer. Non-executable user stack. http://www.openwall.com/linux/, 2000.

[12] R. Wojtczuk. Defeating solar designer non-executable stack patch, Jan 1998. http://www.insecure.org/sploits/ non-executable.stack.problems.html.

[13] Pax. https://pageexec.virtualave.net.

[14] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb 2000.

[15] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proc. ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[16] D.Larochelle and D.Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, Aug 2001.

[17] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, June 2003.

[18] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.

[19] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. http://downloads.securityfocus.com/library/ StackGuard.pdf.

[20] Stackshield - A stack smashing technique protection tool for linux. http://www.anglefire.com/sk/stackshield.

[21] Hiroaki Etoh and Kunikazu Yoda. Propolice - Improved stack smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC)*, (14), 2001. http://www.ipsj.or.jp/members/SIGNotes/Eng/ 27/2001/014/article025.html.

[22] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[23] 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.

[24] Kyung suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proc. USENIX Security Symposium*, pages 81–88, 2002.

[25] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard : Protecting pointers from buffer overflow vulnerabilities. In *Proc. USENIX Security Symposium*, 2003.