TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection

A Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Technology

by

Kumar Avijit (Y0170)
Prateek Gupta (Y0240)

under the guidance of
Dr. Deepak Gupta and **Dr. Dheeraj Sanghi**

to the
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur

April, 2004

.

Certificate

.
Certified that the work contained in the report entitled "*TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection*", by Kumar Avijit and Prateek Gupta, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Deepak Gupta	Dr. Dheeraj Sanghi

.

Acknowledgements

We would like to thank our supervisors Dr. Deepak Gupta and Dr. Dheeraj Sanghi, for their guidance and valuable suggestions throughout the course of this project. This project could not have been accomplished if it were not for their insightful guidance, motivation and technical support. Dr. Deepak Gupta also greatly improved the quality of this report by pointing out errors, inconsistencies and typos. We owe our thanks to our batch mates who have cheered us at times and made the project what it is. Finally, and most especially, we are grateful to our family without whom this would not have been possible.

Abstract

Buffer overflow exploits make use of the treatment of strings in C as character arrays rather than first-class objects. The manipulation of arrays as pointers and primitive pointer arithmetic makes 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 report 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 performing bounds checking on variables defined in shared libraries which are loaded at runtime, we follow the same procedure as in the case of the executable. The shared library is first modified with TIED to include information about all global and automatic buffers in the library. This information is then utilized by LibsafePlus to check whether an out-of-bounds address belongs to a shared library or not. 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%.*

Contents

Chapter 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 overflow.

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]. The simplest kind of buffer overflow attack is presented in the following example.

```
void func (char *str) {
   char buffer[16];
   ...
   strcpy (buffer, str);
}
```

Figure 1.1: A simple stack smashing attack

Assuming that the attacker can control the contents of the string `str`, the local array `buffer` on the stack can be overflowed. The return address stored below `buffer` can be modified so that when the function `func` returns, the control jumps to the attack code which itself can be inserted into the program using `str`. Another class of attacks is *return-into-libc attacks* in which the return address is overwritten with the address of an existing C library function such as `system()` instead of pointing to the attacker's code.

Another simple program vulnerable to buffer overflow attacks is shown in Figure 1.2. The given program provides a " write anything anywhere" primitive. The attacker controls the argument `argv` overflowing the character array `buf` during the first `strcpy`. This causes `ptr` to be overwritten. Thus `ptr` can be made to point to any memory location. The second `strcpy` then copies the contents of `buf` to the desired memory location.

```
int main(int argc, char **argv){
        char *ptr=malloc(4);
        char buf[4];
        strcpy(buf,argv[1]);
        ...
        strcpy(ptr,buf);
        ...
}
```

Figure 1.2: A write anything anywhere primitive

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 due to one or more of the following reasons.

- They do not protect against all forms of buffer overflow attacks.
- They break existing code.
- They impose too high an overhead to be successfully used with common applications.

An acceptable solution must tackle all of the above problems.

In this report, 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 and contains modules for registration/deregistration of shared libraries which are used by the program. 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 the executable (program binary) and then augments the binary with an additional data structure containing the size information for all buffers in the program. For handling overflows in shared libraries which are used by the program, TIED is used to modify the shared libraries to include information regarding all buffers defined in the shared library. Each shared library is required to register/deregister with LibsafePlus at the time of loading/unloading. The registration process records a pointer to the type information data structure stored in the shared library with LibsafePlus. This information is then utilized by LibsafePlus to range check buffers defined in the executable or in one of the shared libraries 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.

We organize the report as follows: Chapter 2 describes different techniques used to prevent buffer overflow attacks and discuss briefly their limitations and/or drawbacks. We describe our basic approach in Chapter 3 describing how TIED and LibsafePlus can be used together to thwart buffer overflow attacks. Because TIED rewrites ELF executables and shared libraries, we first describe the ELF file format concisely in Chapter 4 in order to allow better understanding of how TIED works. Next, we describe the implementation of TIED and LibsafePlus in Chapter 5. This is followed by a description of performance experiments and results in Chapter 6. We conclude the paper with a description of limitations of the tool in Chapter 7.

Chapter 2

Related work

In this section we shall review some of the work done to protect against buffer overflow attacks, their limitations and drawbacks.

2.1 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 [6], 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 and 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 [7].

PaX [8] 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.

2.2 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.* [9] formulated the detection of buffer overruns as an integer range analysis problem. 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 and it 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 [10] by Larochelle and Evans [11] 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. For checking, an annotated version of the standard C library headers is used. The tool does not detect all buffer overflow vulnerabilities and often generates spurious warnings.

CSSV [12] 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. Although the tool is complete, it is impractical to use for large programs since it requires the declaration of procedure contracts by the programmer. As for other static techniques, the tool can produce false alarms.

2.3 Runtime techniques

StackGuard [13] 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 explored by Richarte [14].

StackShield [15] 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 are presented by Richarte [14].

Propolice [16] 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.* [17] 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 [18] is a compiler extension which employs the notion of *referent* objects. A 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 access 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/array access has to be checked and it is therefore not fit to be used in a production system.

The C Range Error Detector(CRED) [19] 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 [20] 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 [21] by Cowan *et al.* 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.

Chapter 3

Basic approach

The steps in the protection of a program using TIED and LibsafePlus are shown in Figure 3.1. The key idea here is to augment the executable and the shared libraries (linked with the program) with information about the locations and sizes of character buffers. To this end, the program executable 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 program binary. Similarly, for handling overflows in shared libraries which are used in the program we require the shared libraries to be compiled with `-g` option so that they contain information regarding the sizes and types of all buffers in the library. The next step is to rewrite the executable and shared libraries with the required information as an additional data structure in the form of a separate read-only section. This makes the information about buffer sizes available at runtime. The binary rewriting of the executable and the shared libraries is done by TIED. LibsafePlus is implemented as a dynamically loadable library which must be preloaded for every process to be protected. To enable range checking, LibsafePlus provides wrapper functions for vulnerable C library functions. These wrapper functions check the bounds of the destination buffer before performing the actual operation. In addition to these wrapper functions, LibsafePlus contains modules for registration/deregistration of shared libraries at the time of their loading/unloading. This process is required for recording pointers to the type information data structure in the shared library with LibsafePlus. This enables LibsafePlus to range check buffers belonging to shared libraries in addition to the buffers which are defined in the program. 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.

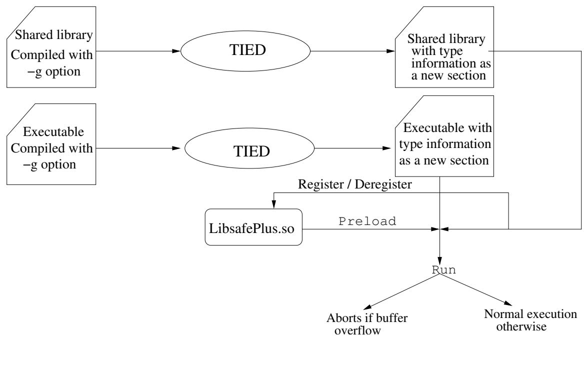
Shared library Compiled with −g option
TIED
Shared library with type information as a new section
Executable Compiled with −g option
TIED
Executable with type information as a new section
Register / Deregister
LibsafePlus.so
Preload
Run
Aborts if buffer overflow
Normal execution otherwise

Figure 3.1: Using TIED and LibsafePlus for buffer overflow protection

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 the buffer instead of determining the exact size. Therefore, it is possible for the attacker to change other variables in the program including local variables and function pointers. 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 provided by Wilander and Kamkar for testing tools for dynamic overflow attacks [22], 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 extension to it. We first describe in Section 3.1, the protection mechanism adopted by Libsafe and go on to show in Section 3.2, how LibsafePlus extends the basic protection mechanism, to handle all forms of buffer overflow attacks.

3.1 Runtime range checking by Libsafe

To ensure that stack smashing attacks do not occur, Libsafe provides a transparent runtime protection system based on the observation that to execute a successful stack smashing attack, the attacker must be able to alter the control flow of the program. Simply copying the attack code into the running program does not lead to a successful attack.

Based on this observation, Libsafe tries to protect from overwriting of frame pointers and the return addresses on the stack. 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 corresponding frame for 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 technique, Libsafe is implemented as a dynamically loadable library which must be preloaded for every process it protects. The preloading is necessary because it injects the library between the program code and the dynamically loadable standard C library functions. The library can then intercept and bounds check the arguments before allowing the standard C library functions to execute. In particular Libsafe provides the following guarantees:

- Correct programs will execute correctly, *i.e.* no false positives.
- The frame pointers and more importantly return addresses can never be overwritten by intercepted function. An overflow that would lead to overwriting the return address is always detected.

Libsafe provides wrapper functions for unsafe C functions like `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. For example, if the user program calls `strcpy()`, the wrapper implemented in Libsafe gets executed - that is due to the order in which the libraries were loaded. The Libsafe implementation of the `strcpy()` function first computes the length of the source string and the upper bound on the size of the destination buffer. It then verifies that the length of the source string is less than the bound on the destination buffer. If the verification succeeds then the `strcpy()` calls `memcpy()` (implemented in the standard C library) to perform the operation. However, if the verification fails `strcpy()` creates a `syslog()` entry and terminates the program. A similar approach is applied to the other unsafe functions in the standard C library.

The Libsafe library has been implemented on Linux. It uses the preload feature of dynamically loadable libraries to automatically and transparently load with processes it needs to protect. In essence it can be used in one of two ways: (1)by defining the environment variable LD_PRELOAD, or (2) by listing the library in /etc/ld.so.preload. The former approach allows per process control where as the latter approach automatically loads the Libsafe library machine-wide.

3.2 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. To perform range checking at runtime LibsafePlus maintains a list of pointers (to the type info structure) for each of the shared libraries and performs a linear search on the list to find out whether a destination address belongs to a shared library. If a match is found, we search the type info structure of the corresponding shared library to find out the bounds of the buffer. If no match is found then we search for the buffer in the type info structure of the program executable. If the type 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.

This chapter presented a brief overview of the basic approach that we adopt to prevent buffer overflows. The actual implementation details of TIED and LibsafePlus are covered in Chapter 5.

Chapter 4

ELF file format

In this chapter, we briefly describe the ELF file format for executable, object and library files. Understanding the ELF file format is necessary to understand how TIED works. ELF [23] stands for Executable and Linkable Format. The ELF format defines a binary interface that is descriptive enough to allow linking of several object files as well as to form a process image during execution. ELF format recognizes the following three kinds of files:

- Relocatable files that can be linked with other relocatable files to form an executable or a shared library.
- Executable files that can be directly loaded into memory and executed.
- Shared libraries that can be linked with other relocatable files and shared libraries. This linking happens in two stages. First, the link editor, *e.g.*, `ld`, processes the shared library with other relocatable files to create an executable. Secondly, the dynamic loader, *e.g.*, `ld.so`, combines it with the executable to form the process image.

The ELF specification provides two parallel views of the object files: *linking* view and *execution* view. The linking view is needed to build the program while the execution view is required to form the process image. To support linking, the linking view divides the file contents into sections that contain the code, data and other useful information for linking such as symbol tables and relocation tables. The execution view, on the other hand describes how various portions of the file should be mapped onto memory while forming the image of the process. It divides the file contents into segments that have permissions like read/write attached to them. Both these views are present in the same file and the ELF header provides a roadmap to describe them. The ELF header forms the first 52 bytes of the file and contains information such as file type, position of section header table in the file, position of program header table, the beginning of executable code, and a 16 byte magic number for quick identification of ELF files.

We next describe the linking view and the execution view in detail.

4.1 ELF linking view

The linking view divides all the contents of the file, except the ELF header, the program header table and the section header table, into various sections. These sections satisfy the following conditions:

- Every section must have exactly one section header in the section header table. The section header table may however contain an entry that does not correspond to any section.
- Every section occupies one contiguous block of memory, possibly of zero byte length.

- No two sections overlap.
- The sections together with the header tables and the ELF header may not cover all the space in the object file.

The section header table contains an entry for every section describing the following attributes:

- **Name** The name of the section. This is actually an offset in a string table that contains all the names.
- **Type** The section type describes the section contents. Sections with special semantic value have special types.
- **Flags** The flags of the section describe if the contents of the section are writable, allocatable, or executable.
- **Address** If the section is allocatable, *i.e.* will appear in the process image, this determines the virtual address at which the first byte of the section will be loaded.
- **Offset** This gives the byte offset from the beginning of the file to the first byte in the section.
- **Size** This gives the number of bytes the section occupies in the file. For special sections like `.bss` which have a type `SHT_NOBITS` and have no content in the file, this attribute gives the number of bytes that the section will have in the process image.
- **Link** This attribute holds a section header table index link whose interpretation is section specific. Some sections like the symbol table are associated with other sections like the string table. As there may be multiple string tables, the link attribute of the symbol table section is used to find out the corresponding string table.
- **Info** This member holds extra information like section header table indices, whose interpretation is section specific.
- **Address alignment** This attribute describes the alignment of the section in bytes.
- **Entry size** Some sections are organized as tables of fixed length records. This attribute gives the size of each entry in such sections.

4.2 ELF execution view

All the ELF files ultimately represent some code to be executed and some data to be used by that code. To execute the ELF file, the operating system first "loads" the program image into memory. The execution view of the ELF object contains control information for construction of this process image. At runtime, the process image is made up of segments of memory that hold the code, data, stack etc. Each segment of the file corresponds to a segment in the virtual address space. A segment holds one or more sections. To understand how sections and segments serve separate needs, consider the following example: A typical executable file contains `.text` section to hold code, `.data` section to hold initialized global data and `.bss` section to hold uninitialized global data. The `.text` section and `.data` section are contained inside two different *segments*. This is because of different access permissions required for text and data: while the data portion must be writable, the text section may not be writable. Because of their similar access permissions, `.data` and `.bss` sections may be part of the same program segment.

Information about how to form the segments in memory is contained in the program header table of the ELF file. The program header table is an array of entries each of which describes a segment. The attributes associated with a program segment that are described by a program header are as follows:

- **Type** The various types of program segments are enumerated below:
  - PT_NULL: Unused entry in the header table.

  - PT_LOAD: This type of an entry describes a segment that must be loaded in the memory.
  - PT_DYNAMIC: This entry specifies dynamic linking information.
  - PT_INTERP: This entry gives the location of the name of program interpreter stored as a null-terminated string in the file. While constructing the process image, the operating system first maps segments from the file to a virtual address space and then invokes the program interpreter which in turn does operations like dynamic linking.
  - PT_PHDR: This entry gives the location and size of the program header table itself. This entry is present only if the program header table is loaded at runtime. If present, this entry must precede any loadable segment entry. A program header table can have at most one entry of this type.

- **Offset** This gives the offset within the file, of the first byte of that segment.
- **Virtual address** In case of executable files, this gives the virtual address at which the segment must be loaded in memory. For shared libraries, the loading is a bit different. A library may be loaded at different virtual addresses in process images of different executables. Therefore, this attribute in the program header specifies the virtual address of the segment assuming that the library is loaded at virtual address 0. The base address of the library at runtime is added to this attribute to obtain the actual virtual address.
- **Physical address** On modern systems, this is same as the virtual address.
- **File size** This specifies the number of bytes of the segment's contents present in the file.
- **Memory size** This specifies the number of bytes in the memory image of the segment.
- **Flags** This attribute holds the access permissions of the segment.
- **Alignment** This specifies the byte alignment of the segment as a power of two.

Figure 4.1 shows linking and execution view of a typical ELF file.

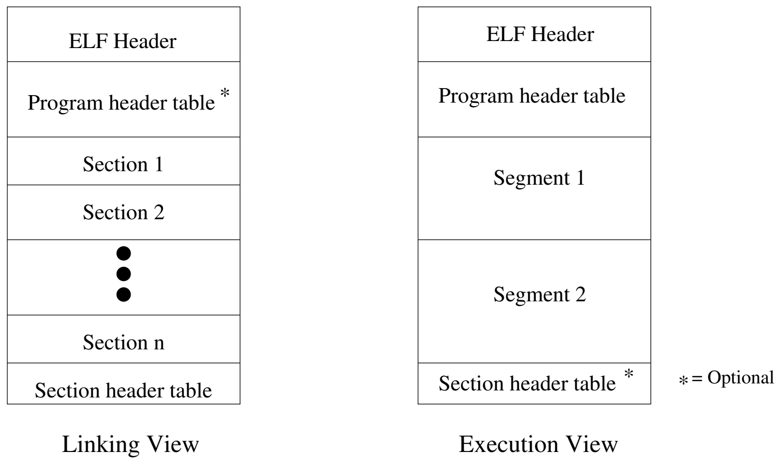

Figure 4.1: Linking and execution views of an ELF file

Next, we describe some special sections and how they contribute in linking an ELF file with another. We categorize the sections based on whether they contain executable code, or program data to be used by code, or control information required to provide the linking view. Code resides in the following three sections:

- **.text** It contains the main executable instructions.

- **.init** This section contains executable instructions that contribute to initialization of the process image. When a program starts to run, the code in this section of the executable is executed before the control passes to the main entry point of the program,*i.e.* `main` function for C programs. In addition, the code in `.init` sections of all the dynamically loadable libraries that are linked to the executable is also executed before passing control to `main`.
- **.fini** Like `.init` section, it contains the code to be executed when the program exits normally.

Data that is used by the code in above sections normally resides in the following sections:

- **.data** This section contains initialized global data.
- **.bss** This section contains uninitialized global data. This sections does not occupy any space in the file. The contents of the section are set to zero while forming the process image.

A number of sections are used for linking of the program. Linking two object files basically involves resolving the symbols that are defined in one object file and used in another. The symbols may either be functions or just global data. To support linking, ELF files have a special symbol table that contains a list of all the symbols used or globally defined in the file. There are typically two symbol tables *viz.* `.dynsym` and `.symtab`. `.dynsym` is used exclusively by the dynamic linker while `.symtab` contains entries for static linking. Each symbol entry in the symbol table describes the following about that symbol:

- **Name** This holds the offset of the name of the symbol in another section called `.dynstr`. The `.dynstr` section just contains a collection of null-terminated strings.
- **Value** The value of a symbol has slightly different interpretations depending on the context. In relocatable files, it is an offset from the beginning of the section that the object is contained in. For shared libraries and executable files, the value of a symbol represents its virtual address. The value of external symbols is 0 as the virtual address is not known until loading.
- **Size** This specifies the number of bytes contained in the object described by the symbol. The size is set to 0 if the symbol has no size or an unknown size.
- **Info** This specifies the symbol's type and binding attributes. The type of a symbol classifies the object that is represented by the symbol, *e.g.* a function, a data object, a section etc. The binding attribute of a symbol describes the visibility of the symbol within other files. A symbol may either be `local`, in which case it is not visible outside the file where it is defined, or `global`, in which case it is visible in all other files. Besides these two bindings, a symbol may also be `weak`. `Weak` symbols have the same visibility as `global` symbols but have a lower precedence.
- **Section index** This gives the index of the section in the section header table, that contains the object represented by the symbol.

The process of connecting symbol reference with their definitions is called *relocation*. For example, a call to a function defined in a library from an object file must somehow transfer the control to the executable code of that function in the library. Relocation is needed because the virtual address of all symbols is not available at runtime. The virtual address of any function or data item inside a shared library is not known statically. To use such a variable, a global offset table is used that, at runtime, will contain the addresses of all the externally defined symbols used in a program. Thus, to use a variable that is defined in say a shared library, the code first accesses the `got` entry of that variable. This gives the address of the variable which can then be accessed. A relocation entry instructs the dynamic loader to relocate the data at a specific location with respect to a symbol. It becomes the responsibility of the dynamic loader to figure out the address of the symbol by searching through symbol definitions in all the files loaded, and perform the specified relocation. Thus, in case of a `got`, the relocation entry specifies the symbol whose address is to be written to a location inside `.got`. A relocation entry contains the following information:

- **Offset** The offset of the location that needs relocation.

- **Info** This gives both the symbol with respect to which the relocation must be made, as well the type of relocation to be performed.

We list below the relocation types used for dynamic linking(we omit the types used by the static linker):

- R_386_RELATIVE: The offset specifies a location that contains a relative address. The dynamic linker adds to this address, the base address at which the file was loaded, to get the absolute virtual address.
- R_386_GLOB_DAT: An entry of this type instructs the dynamic linker to find out the address of the symbol specified in the `info` and write the address at the specified offset.
- R_386_JMP_SLOT: The offset gives the location of a `got` entry associated with a function. This entry instructs the dynamic linker to modify the `got` entry to transfer control to the actual function being called. We elaborate upon this relocation type when we discuss the use of procedure linkage table.
- R_386_COPY: The offset refers to a location in a writable segment. An entry of this type instructs the dynamic linker to search for the symbol in the shared libraries and copy the value contained there to the location specified by offset. The symbol must exist both in the shared library as well in the object file. This type of relocation is used or handling `extern` variables. The following example helps in clarifying the point.

  Consider the following program:

  ```
  extern int extern_var;
  int main(){
      extern_var=2;
      f();
      return 0;
  }
  ```

  The above program is linked with a library that contains the following code:

  ```
  int extern_var=1;
  void f(){
      extern_var=3;
      return;
  }
  ```

  Though both the programs are using the same variable, the library as well as the executable, both have space for the variable `extern_var`. The library has a 4 byte space in `.data` section while the executable has a space in `.bss` section. However, at runtime, only one of the above locations is actually used. The code for the executable simply accesses the location in `.bss` to read/write to `extern_var`. A relocation entry of type `R_386_COPY` instructs the dynamic linker to put the initial value of 1 at the appropriate location in `.bss`. The access to the variable in the library is however a bit complicated. This is because it depends on whether the executable with which the library is linked also references the variable. In case it does not, the library must access the variable directly from its `.data` section. For this purpose, the accesses to the variable are made via a global offset table entry. The `got` entry, at runtime contains the correct value of the address of `extern_var`, *i.e.* a location in `.bss` of executable in the above case, or the location in `.data` of the library in case the executable does not reference `extern_var`. This `got` entry is relocated using a relocation of type `R_386_GLOB_DAT`.

4.3 Calling functions defined in shared libraries

Calling functions that have been defined in shared libraries poses problems similar to those in accessing data items defined in shared libraries. The address of the “external” function is not known until at runtime.

Though this problem can be solved by using `.got` to contain the correct address at runtime, such a solution imposes a performance penalty in terms of having the dynamic linker resolve functions that may not be called from the executable. Such a performance penalty becomes more evident in case of large libraries like `glibc` that contains a huge number of functions. To efficiently solve the problem, calls to "external" functions are made using a procedure linkage table contained in `.plt` section. The idea is to invoke the dynamic linker to resolve a function reference only when the function call is made. A function call, thus transfers control to the code in `.plt` which in turn does the following:

- Invoke the dynamic linker if it is the first call to this function. The dynamic linker resolves the function address and returns control to the function directly. In addition, it fixes the address of the function to be used in later calls.
- In case this is not the first call to this function, the address of the function is already available. Call the function directly.

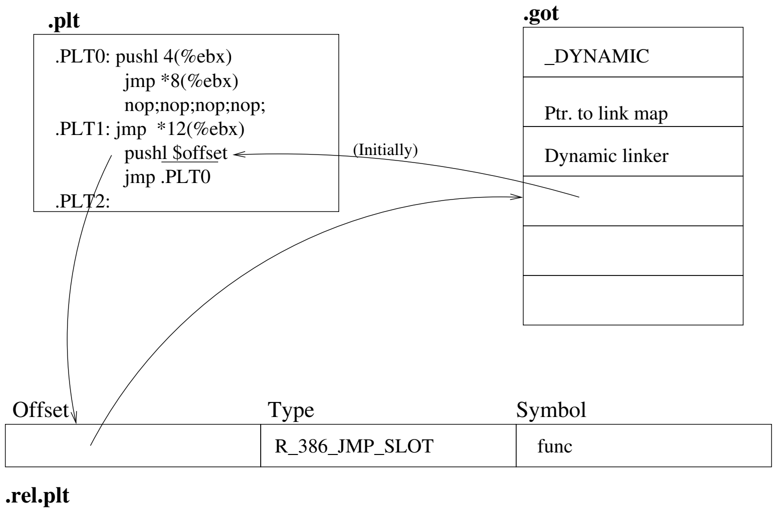

Figure 4.2: Calling an externally defined function using .plt

Now, we describe how the `plt` code invokes the dynamic linker or the function. Figure 4.2 shows a typical `plt` entry for a function `func` and describes how `.plt`, `.got` and `.rel.plt` sections contribute to making a function call. `.rel.plt` is a relocation section that handles `plt` specific relocations.

Before calling the `plt` entry, the register `ebx` is set to the address of `.got`. The first three entries of the `.got` hold special values. The first entry points to the dynamic structure, *i.e.* the `.dynamic` section. This is because the dynamic linker needs to locate its own dynamic structure without having yet processed its relocation entries. The second `.got` entry points to the link map of the object at runtime. The link map of an ELF object is like a handle for that object available to the dynamic linker. The link map contains information like the base address at which the object is loaded, references to the dynamic structure and the symbol tables and hash tables for symbol lookup, and pointers to link maps of other objects that this object depends on. There is loads of other interesting information in the link map but we shall not delve any deeper in this. This entry is initially 0 and is set by the dynamic loader before transferring control to the program. The third entry contains the address of the function `_dl_runtime_fixup` of the dynamic linker. This function is used to relocate data at runtime. This is also set by the dynamic linker before transferring control to the program initially. The `.got` entry corresponding to the function `func` initially points to the instruction: `pushl $offset` in the plt entry for `func`. When the plt code gets executed for the first time, it pushes the offset of relocation entry in `.rel.plt` corresponding to `func` on the stack. Then it pushes a pointer to the link map and calls the dynamic linker. The dynamic linker unwinds the stack and processes the relocation request which is of type `R_386_JMP_SLOT`. The relocation offset points to the `got` entry corresponding to `func`. The dynamic linker searches for the function `func` in other objects defined in the link map and puts the correct address in the `.got` entry. Then, it directly passes control to the required function. As the `.got` entry is fixed, further calls to `func` need not go through the dynamic linker.

4.4 The dynamic structure

Every ELF file participating in dynamic linking has a section called `.dynamic`. This section contains useful information for the dynamic linker. The `.dynamic` section contains an array of (tag,value) pairs. We summarize below, some of the tags and values used in the dynamic section.

- **DT_NEEDED** This element holds the string table offset of a string giving the name of a library needed for linking. There may be multiple number of such entries in a dynamic structure.
- **DT_PLTRELSZ** This element holds the total size of the relocation entries associated with the procedure linkage table.
- **DT_PLTGOT** This element holds the address of `.plt` and/or `.got`. The actual contents are processor specific.
- **DT_HASH** This element holds the address of the symbol hash table.
- **DT_STRTAB** This element holds the address of the string table.
- **DT_SYMTAB** This element holds the address of the symbol table.
- **DT_STRSZ** This element holds the total size of string table.
- **DT_REL** This element holds the address of relocation table.
- **DT_RELSZ** This element holds the size of relocation table.
- **DT_RELENT** This holds the size of each entry in the relocation entry.
- **DT_INIT** This holds the address of initialization function.
- **DT_FINI** This holds the address of the termination function.
- **DT_JMPREL** This holds the address of relocation entries associated solely with the procedure linkage table.
- **DT_VERSYM** This points to a section containing version numbers of all the symbols in `.dynsym`. We describe the symbol versioning concept in detail in Section 4.6.
- **DT_VERNEED** This points to the section containing required version numbers of external symbols.
- **DT_VERNEEDNUM** This element specifies the number of entries in the section of type `VERNEED`.
- **DT_VERDEF** This points to the section containing version numbers of symbols defined in the file.

4.5 Hashing the strings for easy access

The dynamic linker frequently needs to search for symbols inside the symbol tables of files. To speedup the lookup, a hash table of all symbols in `.dynsym` that are strings, is constructed. The hash table directly gives the offset of the symbol in `.dynsym` section. Figure 4.3 specifies the hash function used to hash a string.

The symbol hash table is organized as shown in Figure 4.4. The bucket array contains `nbuckets` entries and the chain array contains `nchains` entries. Both `bucket` and `chain` hold symbol table indices. Suppose

```
unsigned long elf_hash(unsigned char *name){
       unsigned long h=0,g;
       while(*name){
                h = (h<<4) + *name++;
                if( g = h&0xf0000000 )
                        h^= g >>24;
                h  &= !g;
        }
        return h;
}
```

Figure 4.3: The ELF hash algorithm

the ELF hash of a symbol name returns `x`. Then the index `bucket[x%nbuckets]` in the symbol table holds the desired symbol. In case of a collision however, the index may hold a different symbol that hashes to the same bucket. In such a case, the linked list of all symbols hashing to the same bucket can be traversed by using the chain starting at `chain[x%nchains]`.

nbuckets
nchains
bucket[0]
*
*
bucket[nbuckets - 1]
chain[0]
*
*
chain[nchains - 1]

Figure 4.4: Structure of ELF hash table

4.6 Symbol versioning

The concept of symbol versioning [24] has been used to allow a shared library to contain multiple incompatible definitions of the same function. The concept was needed to do away with changing the major number of the library each time a small "incompatible" change was made to some function. "Incompatible" changes include changing the return type or the parameters to the function. This is because during linking, only the major number of the library is matched. This effectively means that an incompatible change cannot be incorporated in a library without changing the major number. Symbol versioning solves this problem by allowing multiple definitions of a symbol to coexist using different version numbers in a library. An old and a new application may now use the same library but different versions of the functions.

An ELF file may or may not use symbol versioning. In case it does, all the symbols in `.dynsym` have a version identifier. In case of a symbol required by the file, the version identifier is matched while searching for the symbol defined in shared libraries. The symbol versioning is implemented using the following three sections:

- **.gnu.version** This section has the same number of entries as the dynamic symbol table. Each entry specifies the version defined or required by the corresponding symbol in `.dynsym`. For symbols that are defined in the file, the value is `vd_ndx` member of a `Verdef` entry in the section `.gnu.version_d`. For symbols that are "imported" from other files, the value is `vna_other` member of the `Elf32_Vernaux` structure present in the section `.gnu.version_r`. Two special values 0 and 1 are reserved and may not appear in `.gnu.version_d` or `.gnu.version_r`. 0 is used for a symbol defined locally that is not available outside the file. 1 is used for a symbol defined in the file that is globally available.

- **.gnu.version_d** This section contains the version identifiers for symbols defined in the file. The section is formed as an array of `verdef` entries. The link of this section in the section header table points to the section that contains the strings referenced by this section.

  ```
  typedef struct {
      Elf32_Half    vd_version; // Version revision. Currently set to 1.
      Elf32_Half    vd_flags;   // Version information flag bitmask.
      Elf32_Half    vd_ndx;     // Version index numeric value.
      Elf32_Half    vd_cnt;     // Number of associated verdaux entries.
      Elf32_Word    vd_hash;    // Version name hash value.
      Elf32_Word    vd_aux;     // Offset of the corresponding entry in the
                                // verdaux array, in bytes.
      Elf32_Word    vd_next;    // Offset to the next verdef entry, in bytes.
  } Elf32_Verdef;
  ```

  The version definition auxiliary entries are defined as follows:

  ```
  typedef struct {
      Elfxx_Word    vda_name; // Offset to the version name string in the
                              // string table.
      Elfxx_Word    vda_next; // Offset to the next verdaux entry.
  } Elfxx_Verdaux;
  ```

- **.gnu.version_r** This section contains the version numbers of the symbols that are "imported" from other files. The section is organized as an array of version needed entries with one or more version needed auxiliary entries in between. The structure of a version needed entry is as follows:

  ```
  typedef struct {
      Elf32_Half    vn_version; // Version revision. Currently set to 1.
      Elf32_Half    vn_cnt;     // Number of associated verneed entries.
      Elf32_Word    vn_file;    // Offset to the filename string in the
                                // section header.
      Elf32_Word    vn_aux;     // Offset to the corresponding entry in
                                // vernaux array.
      Elf32_Word    vn_next;    // Offset to the next verneed entry.
  } Elf32_Verneed;
  ```

  The auxiliary version needed entries are of the following form:

  ```
  typedef struct {
      Elf32_Word    vna_hash;   // Dependency name hash value.
      Elf32_Half    vna_flags;  // Dependency  information flag bitmask.
      Elf32_Half    vna_other;  // Object file version identifier.
      Elf32_Word    vna_name;   // Offset to the dependency name string.
      Elf32_Word    vna_next;   // Offset to the next vernaux entry.
  } Elf32_Vernaux;
  ```

Chapter 5

Implementation

In the following sections we describe in detail the implementation of TIED and LibsafePlus. We shall see in Sections 5.1 and 5.2, how TIED extracts the type information from executable and shared libraries and makes it available as a new loadable section. Finally, in Section 5.3, we describe in detail the implementation of LibsafePlus and how it range checks buffers at runtime by intercepting unsafe C library functions.

5.1 Extracting type information

Type information about variables is present in an ELF file in the form of special debugging sections. DWARF(Debugging With Arbitrary Record Format) [25] has become a standard format to encode symbolic, source level debugging information. Type information in DWARF format is present in the form of DIEs, or Debugging Information Entries. A DIE is the smallest unit of information and may describe a variable, function or a data type. All DIEs have a tag associated with them that describe what the DIEs represent. For example, a DIE describing a variable has a tag `DW_TAG_Variable`. A DIE may have various attributes depending upon the tag. For example, a variable DIE has attributes like address of the variable, the data type of the variable, etc. In addition, DIEs are connected with each other by pointers. For example, a variable DIE will contain a reference to a datatype DIE through a `DW_ATTR_type` attribute. A function DIE has as its children, the DIEs corresponding to all its local variables. Thus the entire type information is accessible by chasing DIE references starting from the top-most function DIE, which is also called the compilation unit.

The gcc compiler generates debugging information in the DWARF format when the `-g` flag is used. TIED uses the *libdwarf* consumer interface [26] to read the DWARF information present in the executable/shared library. For each function in the file, 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 5.1 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 5.1: 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 5.2. 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.

```
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;
```

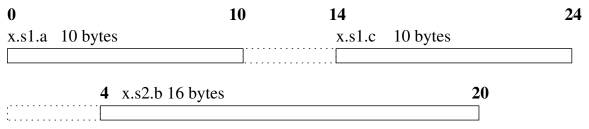

Figure 5.2: Overlapping buffers inside a union

Figure 5.3 describes the approach adopted by TIED in such cases of overlapping buffers. TIED can also

```
Step 1. Arrange all buffers in a list L, in increasing order of their
        starting addresses. Let the buffers be B1, B2, ...
Step 2. If B2 overlaps with B1,
        then
               step 3
        else
               include B1 in the size information and remove it
               from list. Step 4
Step 3. If B2 ends before B1 ends,
        then
               remove B1 from L and make B2 the first element.
        else
               remove B2 from list.
Step 4. Renumber elements as B1, B2, ...
        Repeat step 2
```

Figure 5.3: Conservative choice of buffers inside a union

be made to follow a maximalistic approach by setting a flag, in which case the union of all the overlapping buffers is considered as a single buffer.

5.2 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 ELF file as a new read-only, loadable section. The changes done to the ELF file differ depending on whether the file is an executable or a shared library. We describe both the cases in fuller detail in sections 5.2.1 and 5.2.2.

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 5.4. 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 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.

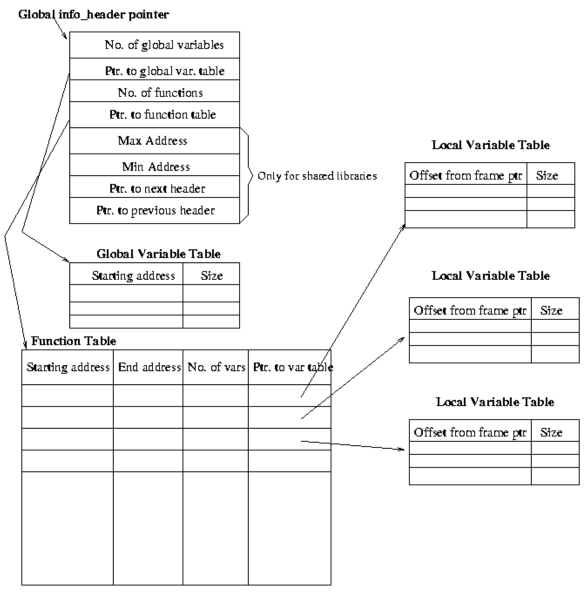

Figure 5.4: Data structures for storing type information

After constructing these tables in its own address space, TIED finds a suitable virtual address in the target file 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.

5.2.1 Rewriting an ELF executable

To place the serialized data structure in the executable, an empty space is first created in the file by extending the executable towards lower addresses by a size that is large enough to hold the type information data structure and is a multiple of page size. This is done because the virtual addresses of sections like

[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.

.text and .data cannot be disturbed in an executable if the code is not position-independent (which is usually the case with executables). The new data structure is dumped in this space in the form of a new section. As the location of the data structure may vary in different executable files, 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 5.5 illustrates the changes made to the target binary.

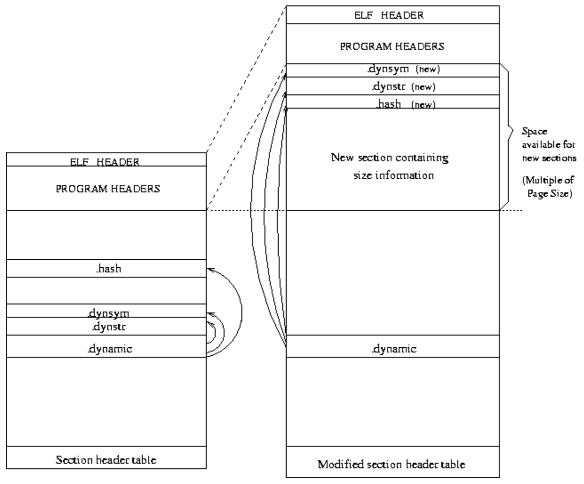

Figure 5.5: ELF executable before and after rewriting

5.2.2 Rewriting an ELF shared library

As compared to rewriting an executable to dump the data structure, rewriting a shared library is more complicated. The main issues in this regard are as follows:

- LibsafePlus needs to know the location of the data structure inside the shared library. Unlike the case of executable, a symbol pointing to the new section can not be used in the case of shared libraries because an executable may depend on multiple libraries. Our approach to solve this problem is based on proactive registration by the shared libraries to LibsafePlus in order to convey the location of the data structure.
- Unlike ELF executables, the virtual addresses of libraries begin from 0 because they can be loaded at any virtual address. This means that TIED can no longer extend the library towards "lower" virtual addresses to accommodate the extra sections. This problem can be easily solved as the shared libraries do not depend on the virtual address they are loaded at (in fact, this is very much needed for their shareability). This allows TIED to alter the virtual addresses of other sections as long as the relative offsets between them are preserved.

Library registration

The registration of a library with LibsafePlus needs to be done before any function/data item of the library is accessed by the executable. For this reason, the registration is done during initialization phase of the library. TIED adds an initialization function to the library that does the following:

- First of all, the initialization function written by TIED calls the original initialization function present in the library. This is done after reconstructing the argument stack that the dynamic linker had prepared before calling the initialization function. The dynamic linker pushes three arguments on stack, which the new initialization function copies on the stack before calling the original initialization function.

- The initialization function then locates the `libsafeplus_registerdl` function. This is the function provided by LibsafePlus for registering libraries. The library uses `dlsym` to determine if LibsafePlus is loaded. `dlsym` always succeeds whenever LibsafePlus is present because LibsafePlus is always loaded before(using LD_PRELOAD) any other dynamically loaded library.
- Once the address of the function `libsafeplus_registerdl` is retrieved, the library calls it after pushing the address of data structure on the stack to form the argument.
- If LibsafePlus is not preloaded, the `dlsym` returns 0. In such a case, the initialization function simply returns.

The `type_info_header` of the data structure in shared libraries has the following extra fields besides those in the header of executable files:

- **Minimum address** This holds the base address of the library at runtime. In the file, this entry is set to 0 and is relocated to the correct base address using a relocation of type `R_386_RELATIVE`.
- **Maximum address** This holds the highest virtual address of any byte of the library. In the file, this entry is set to the offset of the last byte of the last loadable segment and is relocated like the `minimum address`.
- **Pointer to the next type_info_structure** and **Pointer to the previous type_info_structure** These two pointers are provided for LibsafePlus to maintain a linked list of `type_info_structures`. During registration, LibsafePlus adds the registered `type_info_structure` node to the linked list.

To arrange calling of the new initialization function at library startup, the `DT_INIT` entry in the dynamic structure of the library is altered to point to the new initialization function.

As libraries can be dynamically loaded and unloaded during the execution of the program (using `dlopen`), there is a need to deregister the library when it is unloaded from the program image (using `dlcose`). The deregistering simply instructs LibsafePlus to remove information about location of the library's data structure. Deregistering happens in much the same manner as registering. TIED adds a new finalizer function that calls the original `_fini` function before deregistering the library with LibsafePlus. LibsafePlus deletes the deregistered `type_info_structure` from the linked list. The `DT_FINI` entry in the dynamic structure is changed to point to the new finalizer function.

The code for registering as well as deregistering is written in the library as a new read-only section. The new initializer and finalizer functions themselves use position-independent tricks to determine the address of data structure at runtime. This is needed because the address of the data structure cannot be hardcoded into code because the base address of the library is not fixed until at runtime.

The initializer and finalizer functions both call `dlsym` to obtain the address of (de)registering functions of LibsafePlus. Normally, function calls to externally defined functions take place via call to the `plt` entry for that function as described in Section 4.3. To avoid adding a new `plt` entry, we resort to a simpler technique for calling `dlsym`. A 4-byte space in a new writable section is reserved for holding the address of `dlsym`. A relocation entry is added to the `.rel.dyn` section for writing the address of `dlsym` in the 4-byte space. This ensures that before control is passed to the initializer function by the dynamic linker, the address of dlsym in the 4-byte location is already fixed by the dynamic linker. As the symbol `dlsym` is added in the `.dynsym` section, the `.gnu.version` and `.gnu.version_r` are modified to have a version number for it. TIED uses `GLIBC_2.0` as the default version identifier for `dlsym` but this name is configurable by the user. In addition, due to `dlsym`, the library becomes dependent on the library `libdl.so`. To reflect this, an entry of type `DT_NEEDED` is added in the dynamic structure of the library.

Relocating pointers in the data structure

The data structure contains two kinds of pointers: the addresses of global variables and functions, and the pointers that link various tables. Since the base address of the library is not known before runtime, the pointers cannot contain absolute addresses. Also, the addresses of global variables and functions as retrieved from the DWARF structures are with respect to a base address of 0. To offer correct view of addresses to LibsafePlus, relocation entries of the type `R_386_RELATIVE` are made for all the pointers. The initial values contained in them is just an appropriate offset which will be added to the base address of the library at runtime during relocation.

Relocation mismatch due to change in address

Addition of new sections leads to an increment in offsets and virtual addresses of existing sections. All the new sections are added at offsets lesser than those of any existing sections. The relocation entries in the sections `.rel.dyn` and `.rel.plt` access the data to be relocated using offsets. These offsets are incremented by TIED to make them point to the correct data. Also, for entries of type `R_386_RELATIVE`, the addendum present at the specified offset is used by the dynamic linker to calculate the correct address after adding it with the base address. This addendum is also incremented by TIED to make it point to the intended data item. In addition, the values of symbols in `.dynsym` also change due to change in virtual addresses. TIED increments the values of these symbols by the amount of extra space added.

Putting all the sections together

A lot of sections are changed by TIED in the process shown above. For all the sections that are increased in size, TIED makes a new copy of them. Sections like `.rel.plt` that are altered but do not change in size are modified in place. Two new program segments are added to accommodate the new sections. The first segment is a read-only segment that houses the following new sections:

- Section containing the new initializer and finalizer functions.
- The new `.dynsym` section.
- The new `.dynstr` section. Three new strings *viz.* `dlsym, libdl.so` and `GLIBC_2.0` are added.
- The new hash section.
- The new `.rel.dyn` section. It contains new relocation entries for relocating the pointers and absolute addresses in the data structures.
- The new `.gnu.version` and `.gnu.version_r` sections.

The second program segment is a read-write segment and houses the section containing the data structure. The data structure is placed in a writable segment because it needs to be relocated. In addition to the data structure, this segment also contains the new `.dynamic` section. The dynamic section needs write permissions because the dynamic linker fixes absolute addresses in it. The entry corresponding to the `PT_DYNAMIC` segment in the program header table is changed to point to the new `.dynamic` section.

5.3 LibsafePlus Implementation

As described in Section 3, LibsafePlus is a dynamically loadable library that intercepts unsafe C library functions and range checks buffers at runtime. In addition to providing for wrapper functions for the class of unsafe C library functions LibsafePlus also provides modules for registration and deregistration of shared libraries. The registration and deregistration functions are required for handling overflows in shared libraries.

We have seen earlier how the shared libraries and executables are modified by TIED to include information regarding the types and sizes of all global and automatic buffers in the form of a separate section. This information is utilized by LibsafePlus to perform bounds verification at runtime.

To perform bounds verification for buffers defined in shared libraries TIED modifies the `_init` and _fini functions of the library which are called at the time the shared library gets loaded/unloaded. This provides the added functionality of registration/deregistration of the shared library with LibsafePlus whenever the library gets loaded/unloaded. The library maintains its maximum and minimum address in addition to its type info structure and pointers to the next and previous type info structure. The maximum and the minimum address of the library are fixed by the linker by adding appropriate relocation entries for the same. The library calls `libsafeplus_registerdl` for registration with a pointer to its type info structure as argument. This registration function updates the next and previous type info pointers to point to the appropriate entries. Thus, the registration and the deregistration process helps to maintain a linked list of type info pointers for the registered shared libraries. Similarly, at the time of unloading `libsafeplus_deregisterdl` function is called which removes the element corresponding to the shared library from the linked list. This linked list of objects is used by LibsafePlus to perform bounds verification at runtime. For a given destination address, LibsafePlus first checks whether the given address belongs to a shared library by comparing its address with the corresponding maximum and minimum address of the shared library. If a destination address is found to belong to a shared library then the buffer is bounds checked by searching for the corresponding buffer entry in the type info structure of the shared library. If no match is found then the buffer is assumed to be defined in the program itself and the corresponding entry is retrieved from the type info structure of the program executable. If no information is available for the buffer then LibsafePlus falls back to the checks performed by Libsafe (loose upper bound for stack buffers and no checks for global buffers) in the default case.

The bounds verification is done by means of wrapper functions that 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, it is determined whether the buffer belongs to a shared library or is defined in the program itself (and the corresponding type info structure is retrieved). This is done by simply traversing the list of type info structures of registered shared libraries and checking whether the destination address lies within the maximum and minimum address of the shared library. If the buffers is not defined in any of the shared libraries, it is searched for in the type info structure of the program executable. Next, 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 type info structure, 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 buffers is not found to be on the stack, it is checked whether the buffer is on the heap. To capture the sizes of all dynamically allocated buffers, LibsafePlus 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 [27] is used to maintain the size information about dynamically allocated buffers. The

[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.

tree contains one node for each buffer allocated using `malloc`, `calloc` or `realloc`. On freeing a memory area using `free`, the corresponding node in red-black tree is removed. Memory allocation for nodes in the red-black tree is done by a fast, custom memory allocator that directly uses the `mmap` call to allocate memory.

It is determined whether the buffer is on the heap by comparing its address with the minimum heap address. The minimum heap address is recorded by the `malloc` wrappers and is the address of the chunk allocated by the first call to `malloc`, `calloc` or `realloc`. The buffer is declared to be on the heap only if its address is greater than the minimum heap address. If the buffer is indeed on the heap, 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 of the appropriate type info structure. As in the case of stack buffers, the appropriate type info structure is found by traversing the list of type info structures of shared libraries. 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.

Chapter 6

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 [22]. 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. In all the experiments described below, only the executable was modified using TIED, and no shared libraries were modified.

6.1 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 consisted of 100 global buffers and 100 functions each of which 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 6.1, we found a constant overhead of $0.8\mu s$ 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 $2.2\mu s$ per call to `memcpy()` as shown in Figure 6.2. This includes the $0.9\mu s$ 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 allocated 1000 heap buffers first. Then it 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 was the right most child in the red-black tree. As shown in Figure 6.3, the overhead due to LibsafePlus is $1.6\mu s$ per call to `memcpy()`.

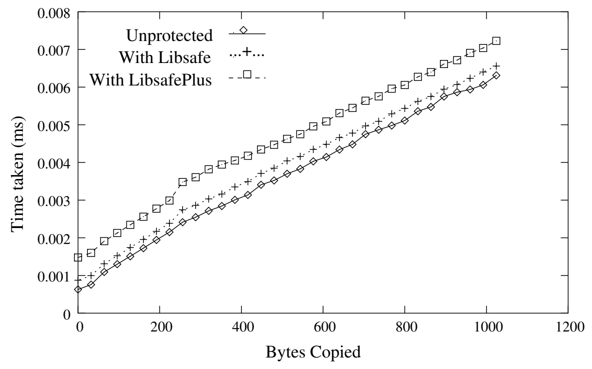

Figure 6.1: memcpy( ) to a global buffer

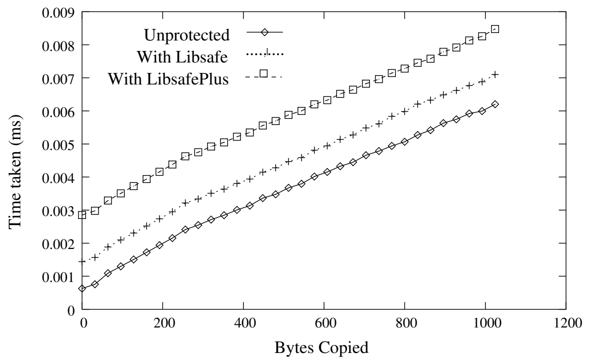

Figure 6.2: memcpy( ) to a local buffer

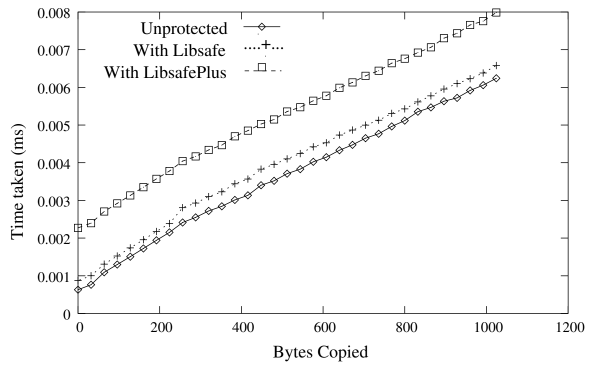

Figure 6.3: memcpy() to a heap buffer

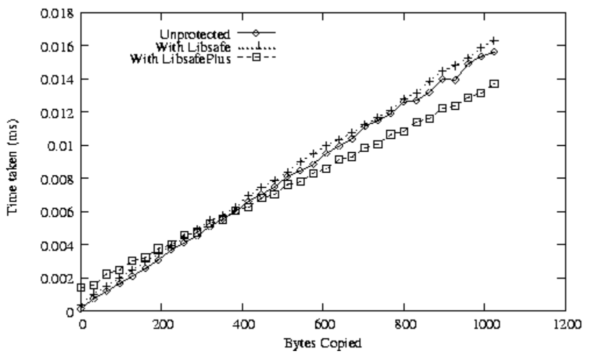

Figure 6.4: strcpy() to a global buffer

We also measured the performance of LibsafePlus for calls to `strcpy()`. The testbed was similar to the one described earlier for `memcpy()`. Figure 6.4 shows the time taken by one `strcpy()` to global buffers. The overhead drops from 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 6.5 and 6.6 demonstrate similar results for `strcpy()` to local and heap buffers respectively.

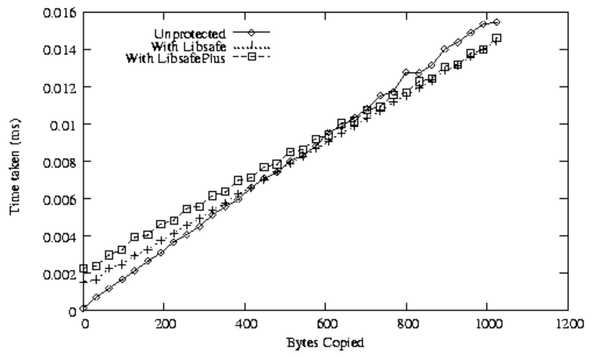

Figure 6.5: strcpy() to a local buffer

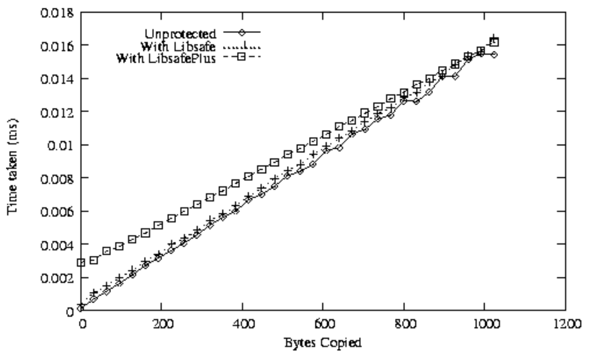

Figure 6.6: strcpy() to a heap buffer

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 6.7, the time taken

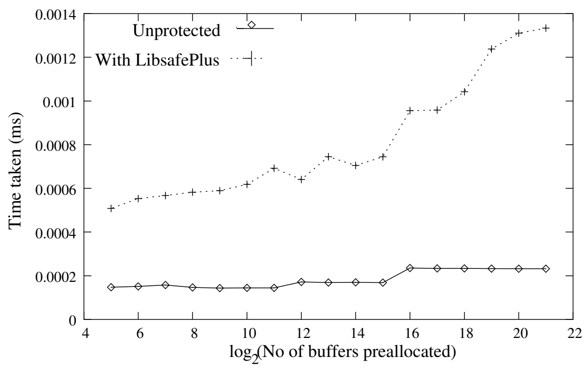

Figure 6.7: Performance overhead for malloc(), free() pair

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.

6.2 Macro benchmarks

Application	**What was measured**
Apache-2.0.48	Connection rate, response time and error rate while requesting large files from the web server.
Sendmail-8.12.10	Time to connect and connection rate achieved while sending large messages.
Bison-1.875	Time to parse large grammar files 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 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 large files 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 6.1: Description of application 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()` to buffers. In all, a total of 12 applications were used to evaluate overhead of LibsafePlus vis-a-vis that of Libsafe. Table 6.1 describes the performance metric used in each case. The performance overheads are shown in Figure 6.8. 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 due to 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 overheads of our tool with CRED [19] (strings only mode). As shown in Table 6.2, 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

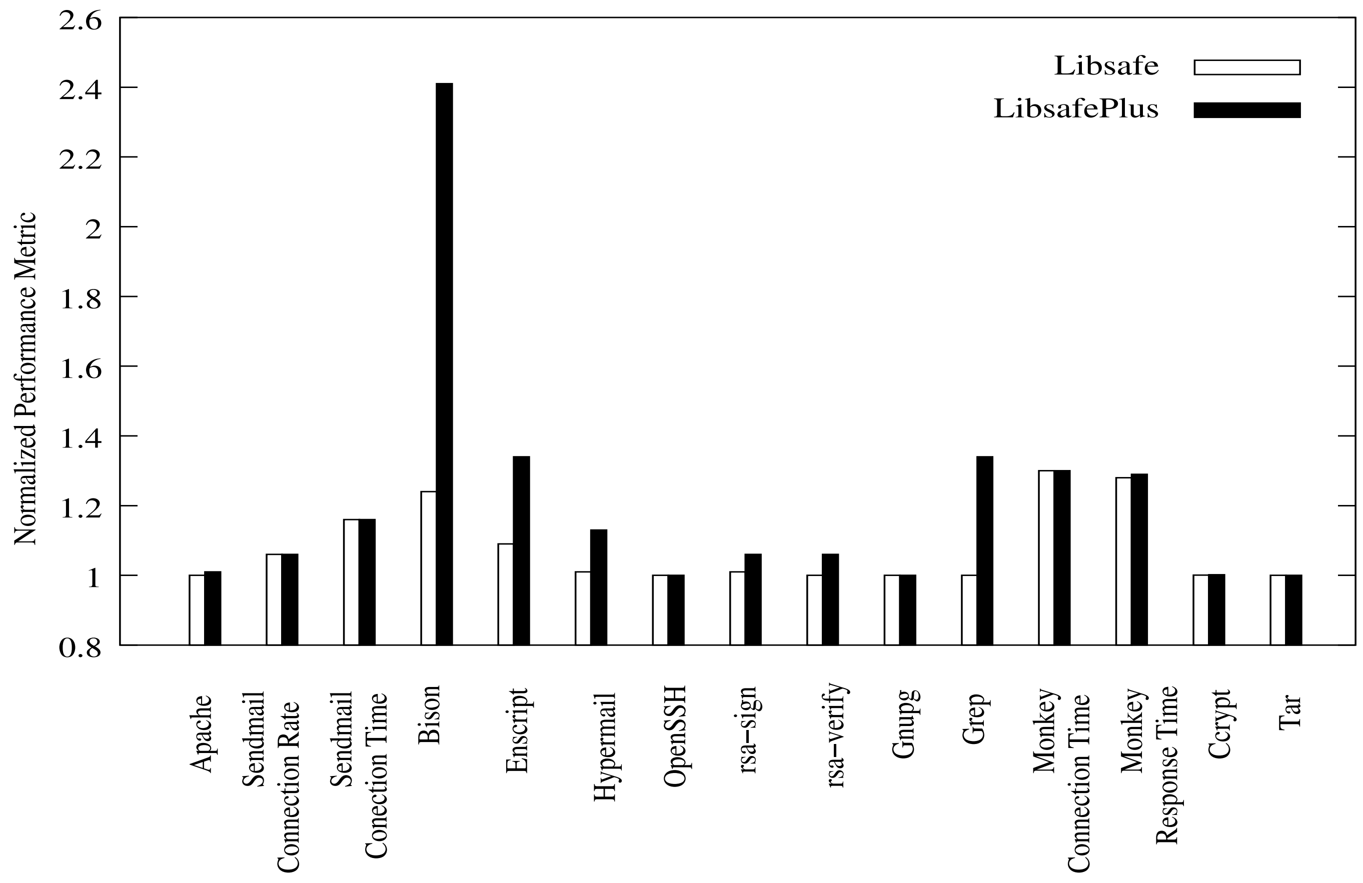

Figure 6.8: Macro performance overheads

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

compared to LibsafePlus, is significant for Apache, Enscript, Hypermail, Gnupg and Monkey.

Chapter 7

Conclusions

In this report, 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.

Bibliography

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proc. 11th Annual Network and Distributed System Security Symposium*, 2003.

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

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

[22] 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.

[23] TIS Committee. Executable and linking format specification version 1.2.

[24] Symbol versioning. http://www.linuxbase.org/spec/book/ELF-generic/ELF-generic/symversion.html.

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

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

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