Detecting Heap Smashing Attacks Through Fault Containment Wrappers

Christof Fetzer Zhen Xiao
{christof, xiao}@research.att.com
AT&T Labs Research
180 Park Avenue
Florham Park, N.J. 07932

Keywords: fault tolerance, debugging support, system security

Abstract

Buffer overflow attacks are a major cause of security breaches in modern operating systems. Not only are overflows of buffers on the stack a security threat, overflows of buffers kept on the heap can be too. A malicious user might be able to hijack the control flow of a root-privileged program if the user can initiate an overflow of a buffer on the heap when this overflow overwrites a function pointer stored on the heap. This paper presents a fault-containment wrapper which provides effective and efficient protection against heap buffer overflows caused by **C** *library functions. The wrapper intercepts every function call to the* **C** *library that can write to the heap and performs careful boundary checks before it calls the original function. This method is transparent to existing programs and does not require source code modification or recompilation. Experimental results on Linux machines indicate that the performance overhead is small.*

1 Introduction

Buffer overflow attacks are a major cause of security breaches in modern operating systems. Many systems are written in the **C** or **C++** programming languages which are optimized for high performance but provide only limited error checking (see Figure 1). For example, the `strcpy(char *dst, const char *src)` function copies a string pointed to by `src` to the location pointed to by `dst`. However, it is up to the programmer to check whether the destination buffer has sufficient memory space to accommodate the source string. Unfortunately, such checks are often omitted in existing programs. Missing checks could be exploited by attackers to get unauthorized access to a computer.

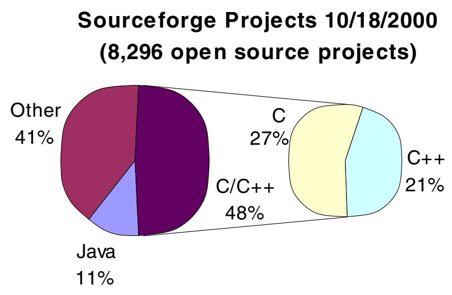


Figure 1. A sampling of www.sourceforge.com indicates that a substantial percentage of open source projects are still using C and C++.

A buffer overflow occurs when a function writes beyond the boundaries of the destination buffer and hence overwrites the content immediately after or before it. Generally this causes a memory corruption or a memory fault of the program. However, it can also be exploited maliciously to alter the control flow of a program in order to break the security of the system. Depending on whether a buffer is allocated on the heap or on the stack one distinguishes between *stack smashing attacks* [5, 2] and *heap smashing attacks*.

A stack smashing attack overwrites a return address stored on the stack to change the control flow of a program. For example, if the return address of a function is stored after a buffer on the stack, a malicious user could overwrite the return address of a root-privileged program by overflowing a preceding buffer and redirect the control flow of the program to a memory region where some attack code is stored. In this way the attacker can gain root privilege.

In contrast, a heap smashing attack overruns a heap

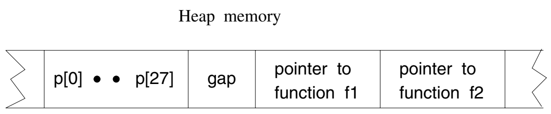
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef int (*f_t)(char*);

void f0(char* p) {
    unsigned char* q = (unsigned char*) &p;
    printf ("last four chars of networkinput should be: \\%o\\%o\\%o\\%o\n",
             q[0], q[1], q[2], q[3]);
}

void f1(char* p) { printf ("f1: %p\n", p); }

char *networkinput = "\150\157\144\145\12\150\143\153\40\143\150\101\164\164
\141\211\341\61\300\260\4\61\333\263\1\61\322\262\14\315\200\303\350\230\4\10";

int main() {
    char* p = malloc (28);
    f_t* f = malloc (sizeof(f_t)*2);
    f[0] = (f_t) f0;
    f[1] = (f_t) f1;
    (*f[0])(p);
    (*f[1])(p);
    strcpy(p, networkinput);
    (*f[0])(p);
}
```

Figure 2. Program with a heap smashing attack: it contains some attack code for Pentium processors in `networkinput`. (`networkinput` has to be modified according to the output of `f1`.)

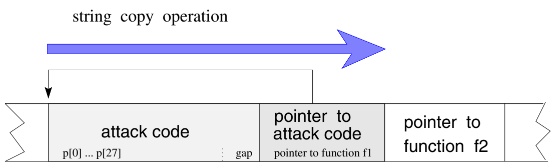
buffer to change the control flow of a program. For example, such an overflow could overwrite function pointers stored on the heap to redirect the control flow of a program. Figure 2 illustrates a situation where a network service with root privilege stores a function pointer `f` after an allocated buffer `p` on the heap. Clients can access this service by sending it network packets which are stored in variable `networkinput`. All packets from legitimate clients are supposed to be at most 28 bytes long. An external attacker, however, may send an oversized packet containing some attack code. When the network service executes the string copy operation to copy `networkinput` to `p`, it would overwrite the function pointer `f[0]` due to a missing boundary check. (Figure 3 compares the memory layout on the heap before and after the `strcpy` call.) Consequently, when this function pointer is called later on, the attacker can gain root access to the computer by hijacking the control of the root-privileged service.

Although less common, heap smashing attacks do occur in practice and can also be dangerous. For example, the recent `bind8` (DNS namer server) attack [11] uses the heap smashing technique to gain root access to a remote computer. Instead of overwriting function pointers, this attack writes malicious shell script code contained in an attack packet on the heap of the server. The attacker gains root access when the script is executed by the `bind8` server.

While safe programming languages like Java are gaining popularity, many projects are still written in unsafe languages like **C** and **C++** (see Figure 1). In these cases, the fundamental solution to buffer overflow attacks relies on a safe coding style: a program should avoid unsafe **C** library functions like `strcpy` and perform careful boundary checks. However, given the huge volume of existing **C** and **C++** programs, it is not possible to inspect and rewrite all of them to eliminate potential buffer overflow problems. Furthermore, while there exist preliminary tools [10] that detect a large class of buffer overruns statically, users might not want to wait until programs are fixed by their developers nor do they have access to the source code of commercial soft-



(a) Before the `strcpy` operation



(b) After the `strcpy` operation

Figure 3. Heap smashing attack as implemented in Figure 2.

ware. A solution that does not need source code access is therefore highly desirable.

This paper presents our work on developing a fault-containment wrapper (**HEALERS**) which effectively protects existing programs from heap smashing attacks caused by **C** library functions. The wrapper intercepts every function call to the **C** library which could be exploited for heap buffer overflows and substitutes it with a version that provides the same functionality but does careful boundary checking. In the previous example, our wrapper would make sure that the `strcpy` function does not access any heap memory outside the allocated buffer `p`. Moreover, our approach is transparent to existing programs and does not require source code access.

The rest of the paper is organized as follows. Section 2 surveys related work in this area. We examine previous work on preventing stack smashing attacks and on developing memory debugging tools for **C** programs. Section 3 describes the details of our technique. We show how the wrapper can keep track of the memory allocation status of a user program, check the boundary of previously allocated buffers, and detect heap smashing attacks. Section 4 evaluates the performance of our method on two Linux platforms. The results demonstrate that the overhead introduced by our wrapper is small. Section 5 concludes this paper.

2 Related Work

Several proposals on preventing stack smashing attacks have been made. StackGuard [3] is an extension to the **C** compiler which prevents such attacks by inserting a *canary* word adjacent to the return address of every function on the stack. Before a function returns, it first checks whether the canary word has been altered. If so, the program assumes that an attack has happened and hence refuses to jump to the location pointed to by the return address. Consequently, an attacker cannot invoke its code by changing the control flow of the program. This approach, however, requires recompilation of the source code and the attacker has to be prevented from reading or guessing the canary word.

Libsafe [2] is a dynamically loadable **C** library which intercepts all unsafe function calls and transparently protects processes against stack smashing attacks. This approach does not eliminate stack buffer overflows. Rather, it confines the scope of any potential buffer overflow to be within the current stack frame and hence protects the return address of the function from being corrupted. Unlike StackGuard, this approach does not require the recompilation of the source code. Our wrapper actually uses the same mechanism as Libsafe to confine stack smashing attacks. Therefore, we do not repeat the discussion in this paper. Unlike Libsafe, our wrapper also prevents heap smashing attacks.

Purify [4] is a software debugging tool that detects memory faults and leaks. It inserts checking instructions for every memory access operation performed by the program. Purify provides comprehensive checking for memory access errors, not just for buffer overflows. In particular, it uses an adaptation of garbage collection techniques to detect memory leaks that usually do not produce directly observable errors. However, to perform these checks it slows down the execution speed of the program by a factor of 2 to 3. While this is not a problem for debugging a faulty program, it is not acceptable if installed as a security protection mechanism on a system-wide basis.

ElectricFence [6] is a public domain heap debugging tool. It uses memory protection techniques to detect heap buffer overflows. It places a read and write protected memory page after each buffer allocated. In this way, a buffer overflow will cause a segment violation. Memory that has been released is also memory protected to detect access to already freed memory. ElectricFence uses two virtual pages per allocated buffer and slows down execution considerably. It is only intended for debugging purposes.

Safe-C [1] uses source-level transformations to make **C** pointers safe. The idea is that additional meta-data is kept for each pointer such that access errors can be detected during run-time. The overhead was reported between 130% and 540%. Our approach keeps the meta-data for each allocated block as opposed to every pointer. It does not need the source code of a program and has an overhead of typically less than 12%. On the other hand, Safe-C can detect buffer overflows that are caused by any function (not necessarily **C** library functions) as well as overflows of buffers that are contained within other buffers.

Vmalloc [8] is a library that extends the malloc function in the Standard **C** library to give applications better control on memory allocation methods. Applications can specify how to obtain new memory from the heap (e.g. best-fit strategy) as well as change the layout of existing memory regions. Unlike other `malloc` implementations, Vmalloc provides clients with the functionality to determine for any given address if this address belongs to an allocated heap block and the start and end address of this block (if it exists). If a system uses Vmalloc, we could simplify our wrapper since we do not need to perform additional state keeping for allocated blocks.

3 Description of the technique

Our goal is to protect existing software against heap smashing attacks caused by **C** library functions even if the source code is unavailable. Furthermore, we want to help software developers by performing most boundary checks automatically for them. We implement our checks as a dynamically loadable **C** library wrapper. The wrapper intercepts every function call to the **C** library which could be used to write to the heap and performs careful boundary checks before it invokes the original function.

Previously, software wrappers have been used for fault-tolerance and exception handling [9, 7] as well as for detecting stack smashing attacks [2]. A nice feature of this approach is that it requires no source code modification or recompilation. On most Unix systems a user interested in using a wrapper can preload it by defining the `LD_PRELOAD` environment variable. This is useful for protecting certain network services. In addition, a system administrator can enable a wrapper on a system wide basis through a dynamic link loader. We have developed and tested our wrapper on Linux systems. However, it can be ported to other systems that provide an appropriate interface to the dynamic link loader. Note that the wrapper only works with programs that are dynamically linked with the **C** library. This is typically no restriction since almost all programs appear to be dynamically linked.

3.1 Keeping track of blocks

In order to perform boundary checks of allocated buffers, we need to keep track of the memory allocation status on the heap. The difficulty is that the existing interface of `malloc` in the standard **C** library is very simple: it only returns the start address of the allocated memory. In order to perform boundary checks, programmers either have to keep track of the allocated blocks or use systems with better debugging support like Vmalloc [8].

Since our goal is to transparently protect existing software even if the source code is unavailable, we use a wrapper to keep track of the allocated blocks and to perform the boundary checks. When a program executes `p=malloc(size)`, it invokes the version in our wrapper which in turn invokes the `malloc` function in the **C** library to perform the memory allocation and then records the position and size of allocated memory in an internal table. Later when the program executes `free(p)`, the wrapper removes the corresponding entry from its table. By doing so the wrapper can be kept up to date with the current memory layout. We describe in Sections 3.3 and 3.4 two alternative algorithms to keep track of memory layout on the heap.

3.2 Boundary Checking

We define two variables `heap_start` and `heap_end` that keep track of the start and the end of the heap. To determine the "range" of a pointer, we implement the following function. Function `range(p)` returns a value < 0 if `p` does not belong to a block allocated on the heap. Otherwise, `range(p)` returns the number of bytes between `p` and the end of the allocated block. Figure 4 illustrates three calls to function `range`.

Using the function `range` and the variables `heap_start` and `heap_end`, we can test if a `strcpy(dst, src)` operation would result in a heap smashing. We have to test that if the address range of the copied string [`dst, dst+strlen(src)+1`][1] overlaps with the address range of the heap [`heap_start, heap_end`], then the range of the destination pointer (`range(dst)`) is longer than the length of *src* (`strlen(src)`). Note that it has to be *strictly* longer to account for the terminating zero. More precisely, we can express this condition as follows. Let s be the length of string src, i.e., `s = strlen(src)`. Calling `strcpy(dst, src)` does not result in a heap smashing only if one of the following conditions is true:

- `dst > heap_end`
- `dst+s+1 < heap_start`
- `s < range(dst)`

However, it still might result in a stack smashing or smashing of other data areas like the static data section.

[1]We need to check for integer arithmetic overflow.

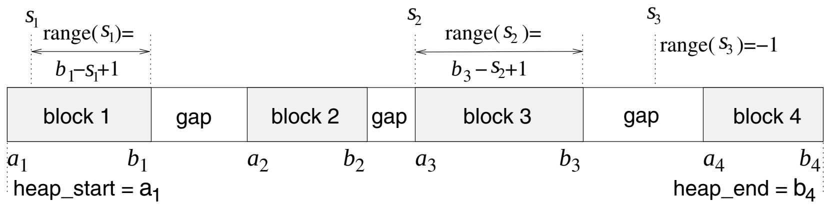


Figure 4. Range of three addresses: s_1, s_2, and s_3.

The wrapper can be configured to evaluate additional conditions that prevent stack smashing attacks. Optionally, it can prevent any smashing caused by **C** library functions by preventing them to write to any location outside the heap and the stack. If a smashing attack is detected, the wrapper logs an error message and aborts the program.

3.3 Red/Black Tree

Our first implementation of `range(p)` searches through all previously allocated memory blocks to find the one that contains `p`. This method, however, requires an efficient algorithm to search through entries in the internal table to locate the appropriate memory block. For efficiency, we organize the entries in the table as a Red/Black tree. Each node in the tree corresponds to an allocated memory block. The key consists of the start and end address of the block. Note that since the addresses of blocks are non-overlapping, there is a total order between these address ranges. It is well-known that the complexity of search, insert, and remove operations in a Red/Black tree is logarithmic to the number of entries in the tree. Hence the protocol is reasonably efficient for large tables.

3.4 Magic Number

Our second implementation of `range(p)` uses a special data structure to reduce the complexity of inserts and removes. In this solution we add an header to each allocated memory block to store certain meta data. The size and the address of each allocated block is kept in a separate table. The amount of memory the wrapper allocates is slightly larger than what is returned to the user as illustrated in Figure 5. One component of the meta data is a *magic* number: a carefully chosen number which is unlikely to appear in ordinary user program. The magic number marks the beginning of a memory block. The second component is an index into the table. The entry of the table contains the address of the allocated block and its size.

Function `range(p)` searches the memory preceding pointer `p` for the magic number. If a word does not match the magic number, then it cannot be the start of a memory block. On the other hand, if a match is found, then it is likely that we found the starting point. However, because a magic number can occur inside a user region coincidentally, the wrapper still needs to consult its table to verify whether the corresponding address is indeed the start of a memory block. (This also makes sure that the magic number was not inserted by a malicious user in an attempt to foul the wrapper.) The wrapper first checks whether the index points to a valid entry in the table. If so, it then compares the address of the magic number and the address stored in the entry to see whether they are the same. If both tests succeed, we know that the start of the allocated buffer is found. The wrapper then computes the range of p by taking the difference between the end address of the allocated block and p (plus 1). This difference is non-negative only if p belongs to an allocated block. The search for the start address ends when either the first allocated block is found or when the search reaches the boundary of the heap (in the latter case it returns -1). We call this a *linear search* algorithm because the amount of search time is linear with respect to the offset of `p` from the start of the memory block (every word could potentially be the starting point).

Ideally, we want to reduce the number of comparisons when searching for the magic number. The number of search steps can be reduced by 50% by aligning the 32-bit magic number on a 64-bit boundary. This permits the wrapper to compare the magic number only with every second 32-bit word. Moreover, this alignment can be done without additional memory overhead: the index can be placed either before or after the magic number to align it appropriately. When a magic number is found, the search algorithm can then test if the word before or after the magic number is a valid index.

We can generalize this optimization to reduce the number of search steps to be logarithmic with respect to the offset of `p`. This is achieved by restricting the start address of every allocated memory block to be a power of 2: let $msb(S)$ be the value of the most signifi-

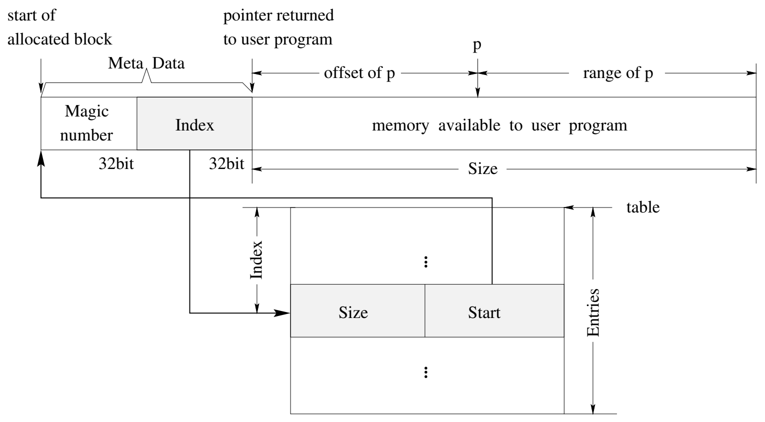


Figure 5. The magic number is used to locate the start of a memory block.

cant bit of the binary representation of S. For example, $msb(22_{10}) = msb(10110_2) = 10000_2 = 16_{10}$. We align the start address of a memory block of size S at a $msb(S)$ byte boundary. This can be achieved through the `memalign` library function or its equivalents. By aligning a memory block at a boundary proportional to its size, the set of addresses which could be the start of the block has been substantially reduced. This in turn reduces the number of search steps to find the magic number: given a pointer, we first check whether the next smaller 32-bit aligned word contains the magic number. If not, we know that the block is bigger than 32-bit. We then check the next smaller 64-bit aligned word. If this also fails, we know that the block is bigger than 64-bit and we can look at 128-bit aligned words, etc. The distance between consecutive words that are checked for the magic number increases exponentially. In effect, we have turned the previously linear search into a *logarithmic search* in that the number of search steps is logarithmic to the offset of the `dst` pointer.

There is, however, a tradeoff between search time and memory usage as well as allocation overhead. Although logarithmic search reduces the amount of search time, it may introduce memory fragmentation and may also increase the allocation time. The implementation of `memalign(alignment, size)` that we use allocates a sufficiently big junk of memory and then frees memory to align the block at an `alignment` byte boundary. Due to the additional work, there is a noticeable performance overhead associated with the `memalign` function call (see Section 4 for details). This raises the question as whether the benefit of the reduction in search time would outweigh the additional overhead due to the `memalign` call. The answer depends on the offsets experienced in string copy operations: linear search is preferable for small offsets due to its low overhead and better memory utilization, while logarithmic search is desired for large offsets because it can substantially reduce the number of search steps.

Ideally, an algorithm should adaptively switch between these two strategies. In fact, for a particular application, it is possible to analyze its memory access pattern and select an appropriate strategy. However, our goal is to develop a generic wrapper suitable for any application. Consequently, we designed a *hybrid search* algorithm: when a program executes `malloc(size)`, the wrapper invokes the original `malloc` function if the size of allocated memory is smaller than a certain threshold *thresh*. Otherwise, the wrapper aligns the allocated memory as described above in order to reduce the amount of search time. When the program later executes `range(dst)`, the wrapper first performs linear search for up to *thresh* steps. If it cannot find the start address of the memory block, then it switches to logarithmic search[2]. This guarantees that the wrapper can have reasonably good performance for all situations. Note that linear search can be viewed as hybrid search with *thresh* = ∞ and logarithmic search as hybrid search with *thresh* = 0.

Table 1 compares the complexity of different solutions discussed in this section. For both linear search and logarithmic search, it takes constant time to insert an entry into the internal table or remove an entry from the table. The advantage of logarithmic search is to reduce the complexity of search operation. Note that hybrid search (with a constant *thresh*) has the same complexity as logarithmic search, which has the best complexity among all three approaches. We investigate in Section 4 if this

[2]Note that we cannot skip the initial linear search phase since we do not know that the block enclosing `dst` is greater than *thresh*. However, one could skip this phase if blocks of size smaller than *thresh* are allocated in a different region than block of size greater than *thresh*.

Algorithm	search	insert	remove
Red/Black tree	O(log(entries))	O(log(entries))	O(log(entries))
linear search	O(offset)	O(1)	O(1)
logarithmic search	O(log(offset))	O(1)	O(1)
hybrid search	O(log(offset))	O(1)	O(1)

Table 1. Complexity of the investigated heap overflow detection methods.

results in some real performance benefits.

3.5 Implementation Issues

As described previously, our wrapper intercepts every **C** library function call which is related to memory operations. When an application program calls `malloc`, for example, it invokes the `malloc` function in our wrapper. The wrapped `malloc` function then needs to resolve the original `malloc` function in the **C** library through an interface function `dlsym` of the dynamic link loader. (This only needs to be done when the wrapped `malloc` is called for the first time.) In some situations, however, the `dlsym` function may actually attempt to allocate memory by calling `malloc`. In UNIX (unlike in Windows), this results in another call to the wrapped `malloc`. If the wrapper again calls `dlsym` to resolve the original `malloc` function, it leads to an indefinite recursion which ultimately ends up with stack overflow.

We address this problem by associating a recursion detection variable with `malloc`. This variable records whether the wrapped `malloc` has been called previously. If so, then the wrapper does not call `dlsym` again to resolve the original `malloc`. Instead it returns an empty pointer. The GNU implementation of `dlsym` that we use in this case uses a static buffer for the resolution of `malloc`. For other implementations of `dlsym` function the wrapped `malloc` could return statically allocated memory until the original `malloc` is resolved.

Another implementation problem we encountered is when an application calls `realloc` to increase the size of a previously allocated block. When the logarithmic or hybrid search approach is used, the alignment boundary of an allocated block depends on its size. Hence increasing the size of a block might require a realignment of the buffer. In this case, our implementation allocates a new block that is appropriately aligned, copies the old user data, and then frees the old block. This is consistent with the semantics of `realloc` which also copies a block in case it cannot resize it at its current address.

4 Performance Measurements

This section evaluates the performance of our wrapper on two different architectures:

- a 450 MHz Pentium II with 256 MByte of memory;
- a 900 MHz Atlon with 784 MByte of memory.

We focus on the performance of four utility programs: *tar*, *gzip*, *gcc*, and *ps2pdf*. They represent a spectrum from I/O bound applications (like *tar*) to CPU bound applications (like *gzip*). We compare the performance of the two solutions proposed in the previous section: using Red/Black tree versus using magic number. In the latter case, we also compare the performance of different search algorithms. In this paper, we mainly present results on the Atlon machine. Results on the Pentium machine are similar and hence omitted due to lack of space.

4.1 Memory Access Patterns

We first present some statistics on the memory access patterns of the four utilities. This is important because the optimal solution for a particular application depends on its memory access patterns. For example, the choice between linear search and logarithmic search depends on the offsets that arise in string copy operations as explained in the previous section. Logarithmic search is efficient in reducing search steps for large offsets, but pays a higher overhead when allocating blocks and also leads to memory fragmentation. This raises the question as how often large offsets occur in practice. To answer this question, we measured the offsets of the following **C** string library functions: `strcpy`, `strncpy`, `strcat`, and `strncat`. While functions `strncpy` and `strncat` are typically assumed to be safe, we also enabled run-time checks for these functions to check that the passed string sizes are accurate. Figure 6 shows the results for the *tar* program. The figure indicates that a substantial fraction of string function calls are with small offsets. For example, about 40% of string copy operations in *tar* are called with an offset 0. However, the remaining 60% of these calls can have an offset of up to 10000 bytes. Similar trends have been observed for other programs we measured. These results suggest that hybrid search is a viable approach both for minimizing allocation overhead for small offsets and for reducing search steps for large offsets.

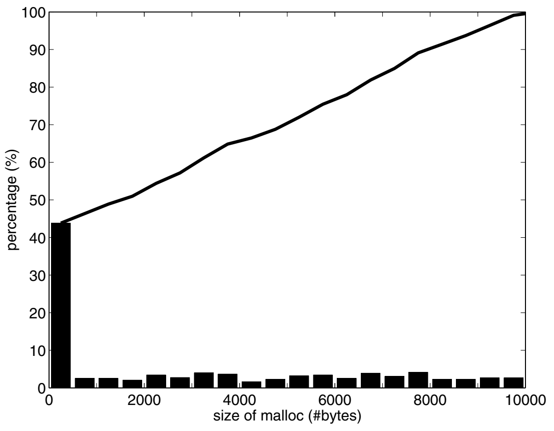


Figure 6. Offsets measured when calling C library string copy functions for the *tar* program. The graphs also contain the cumulative percentage of the offsets measured.

The overhead for wrapping a function is minimal. On the 450MHz Intel Pentium II we measured that inserting an empty wrapper has an execution time overhead of about 76ns. The boundary checks performed by a wrapper is another factor that affects the the performance overhead. Intuitively, the more frequent an application performs boundary checks, the higher the penalty will be. Table 2 shows the functions we wrapped for our measurements and the number of wrapped function calls for the four utilities. For example, the table indicates that *gcc* invokes a large number of `free` calls. Whenever a memory region is freed, the wrapper needs to consult its internal table to verify that the freed pointer is valid. This helps explain some of the observed performance overhead described later in this section.

4.2 Malloc Overhead

Our wrapper needs to update certain data structures whenever a memory block is allocated. In the Red/Black tree solution, each block that is allocated needs to be inserted into the tree. In the hybrid search solution, the wrapper needs to initialize certain meta data at the beginning of the allocated block and update the entries in its internal table accordingly. Moreover, for blocks larger than a certain threshold, the wrapper needs to call the `memalign` function to align the allocated block at an appropriate boundary. In the GNU implementation of `memalign` (glibc-2.1.2), this is achieved by first allocating a sufficiently large block and then freeing a prefix of it. We have found that for large blocks this results in a substantial slow-down with respect to the original `malloc` function of the **C** library. Figure 7 investigates the performance overhead for allocating memory blocks with different block sizes.

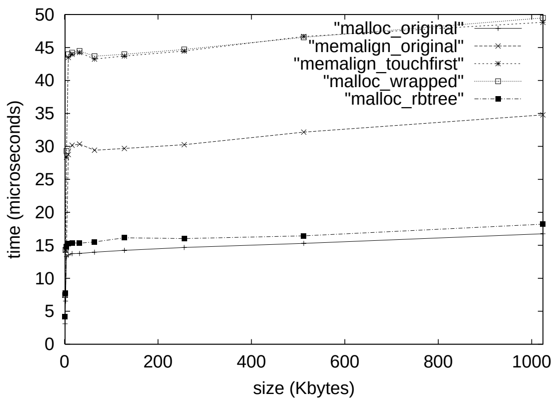


Figure 7. Time to allocate block for hybrid search and Red/Black trees in a 900MHz Atlon (mean of 100 executions trimmed 10%).

As can be seen from the `malloc_original` and `malloc_rbtree` graphs in Figure 7, the wrapper incurs a negligible overhead when using a Red/Black tree, but a noticeable overhead when using magic numbers (graph `malloc_wrapped`). The overhead of using magic numbers for very large blocks can be explained by the fact that in modern operating systems memory pages are allocated on demand: an allocated page is mapped to a physical page when the page is accessed for the first time. For large blocks, `malloc` suffers one page fault because `malloc` inserts meta data before the start of the block (see graph `malloc_original`). Because the implementation of `memalign` splits a block into two blocks, it writes two sets of meta data and hence, suffers two page faults for very large blocks (see graph `memalign_original`). When using magic numbers, our wrapper needs to insert the magic number at the beginning of the allocated block. For large blocks this magic number is written at the start of a page break. This introduces another page fault since the meta data of `malloc` is stored at the end of the preceding page. This explanation is supported by graph `memalign_touchfirst` which displays the time of performing `memalign` plus the time of writing the first byte of the allocated block. Since most applications will touch the allocated memory region, we expect that the

Function	tar	gzip	gcc	ps2pdf
malloc	169 (55.78%)	3 (16.67%)	27618 (46.92%)	9576 (37.27%)
calloc	6 (01.98%)	1 (05.56%)	197 (00.33%)	5 (00.02%)
realloc	0 (00.00%)	0 (00.00%)	38 (00.06%)	0 (00.00%)
free	41 (13.53%)	2 (11.11%)	17418 (29.59%)	9564 (37.22%)
strcpy	30 (09.90%)	1 (05.56%)	8995 (15.28%)	115 (00.45%)
strncpy	44 (14.52%)	1 (05.56%)	3397 (05.77%)	0 (00.00%)
strcat	0 (00.00%)	0 (00.00%)	694 (01.18%)	103 (00.40%)
fileno	6 (01.98%)	4 (22.22%)	5 (00.01%)	0 (00.00%)
fread	0 (00.00%)	0 (00.00%)	475 (00.81%)	6331 (24.64%)
signal	7 (02.31%)	6 (33.33%)	24 (00.04%)	0 (00.00%)
total	303 (100.0%)	18 (100.0%)	58861 (100.0%)	25694 (100.0%)

Table 2. Number of wrapped function calls (some were called by the "measurement" wrapper).

overhead of at least one of the two additional page faults of the wrapped `malloc` will be amortized later on.

4.3 Search Overhead

Before a wrapped function (e.g., `strcpy`) writes into a memory region, it calls `range` to determine whether the destination buffer has sufficient space. For the hybrid search algorithm, the number of comparisons needed by the wrapper depends on the selected threshold *thresh*.

To measure the total search execution time overhead of our algorithms, we extracted the memory allocation and search patterns of the four utilities using a "measurement" wrapper. We used the data to generate micro-benchmarks which call the wrapped functions with the same offsets, i.e., they have the same search behavior as the utilities. The micro-benchmarks execute nothing else than the wrapped functions. We approximate the search time overhead of the four utilities by measuring the difference in the execution times of the micro-benchmarks with and without a wrapper. Figure 8 shows the execution time overhead of hybrid search in a 900MHz Atlon with respect to the threshold.

We used the same micro-benchmarks to compare the search time overhead of Red/Black trees versus logarithmic search for two different architectures and show the results in Figure 9. The reported execution time overhead is the mean of 100 execution time differences trimmed 10%. (The trimming filters noise introduced by the benchmark programs getting dispatched.) The overhead of logarithmic search (i.e., $thresh = 0$) is in most cases better than that of Red/Black trees.

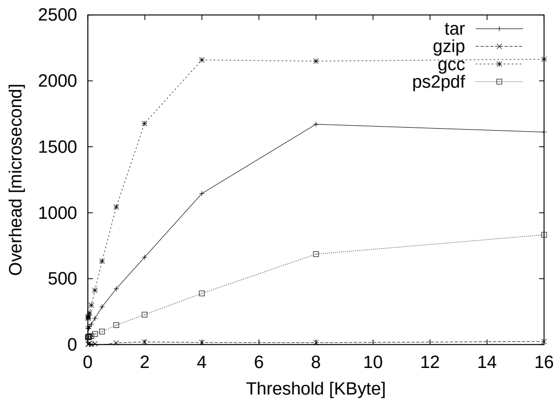


Figure 8. Search time overhead for hybrid search in a 900MHz Atlon.

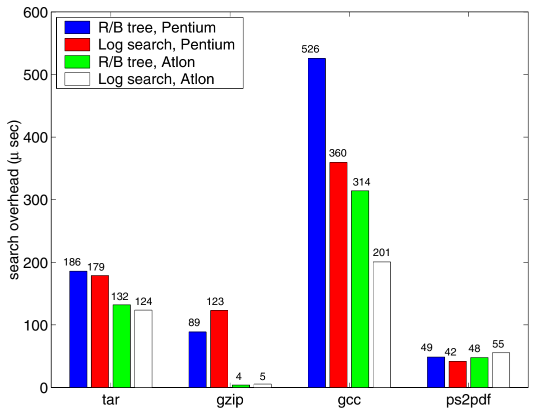


Figure 9. Comparison of search overhead between Red/Black tree and logarithmic search.

4.4 Application Speed

We measured the overall performance overhead of the four utility programs using hybrid search and using the Red/Black tree. The results are shown in Figure 10. The performance overhead can be explained due to the allocation/free overhead, the search time to find the meta-data of an allocated block, the wrapping time of function calls, and the time to preload the wrapper. The difference in the number of wrapped functions called is given in Table 2. Based on our performance measurements, we use hybrid search with a general threshold of 7 bytes in our wrapper. If the behavior of a program is known, using a higher threshold can be advantageous.

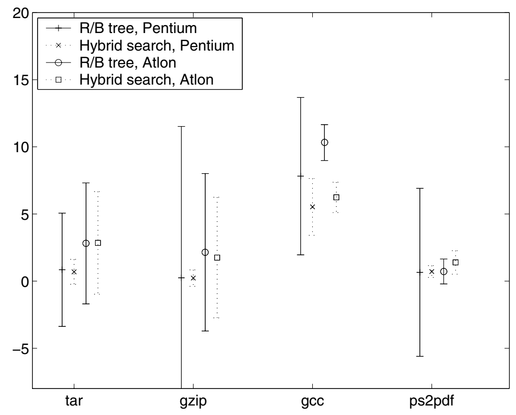


Figure 10. Execution time overhead for two different architectures (mean of 100 executions trimmed 10%).

In summary, our hybrid search wrapper (threshold of 7) is slightly faster than the Red/Black tree wrapper. On the other hand, the Red/Black tree solution has advantages with respect to memory fragmentation and memory allocation time. Ideally, one could use the same Red/Black tree for implementing `malloc` and `range` and in this way avoid the double bookkeeping.

5 Conclusion

Buffer overflow attacks are a major cause of security breach in existing operating systems. We have presented a fault-containment wrapper which provides efficient protection against heap smashing attacks caused by **C** library functions. The wrapper keeps track of the memory allocation status by intercepting memory related **C** library function calls and performs careful boundary checking before calling the original function.

We have investigated two different strategies to find the "range" of a pointer: Red/Black trees and hybrid search. We installed the wrapper on Linux machines and evaluated its performance for several common applications. The wrapper is reasonably efficient for both strategies.

References

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, San Diego, California, June 2000.

[3] Crispin Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[4] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[5] Aleph One. Smashing the stack for fun and profit. In *Phrack Magazine*, 1998.

[6] Bruce Perens. ElectricFence: ftp://ftp.perens.com/pub/ElectricFence/

[7] Frederic Salles, Manuel Rodriguez, Jean-Charles Fabre, and Jean Arlat. Metakernels and fault containment wrappers. In *International Symposium on Fault-Tolerant Computing*, 1999.

[8] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. In *Software Practice and Experience*, March 1996.

[9] Kiem-Phong Vo and Yi-Min Wang. Xept: A software instrumentation method for exception handling. In *the Eighth International Symposium on Software Reliability Engineering*, 1997.

[10] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2000.

[11] http://www.securityfocus.com/˜vdb/˜-bottom.html?vid=2302