ContraPolice: a libc Extension for Protecting Applications from Heap-Smashing Attacks

Andreas Krennmair
krennmair@acm.org

November 28, 2003

Contents

1 Introduction

1.1 Buffer Overflows

In today's computer security, buffer overflows are a huge problem – most of the time caused by inexperienced programmers using inadequate language without fully understanding all consequences of using. One of these languages that cause such problems ist *C*, an imperative programming language developed in the early 1970s by Dennis Ritchie. Unfortunately, at this time hardly anybody of the big problems that insecure buffer handling could cause. This led to a set of insecure library functions in the first versions of the C standard library that eventually got standardized together with the C language and that are still widely used by inexperienced C programmers – and even by their teachers, since that was the way they learned it themselves. And to keep up backward compatibility, even new versions of the C standard (i.e. *C99*) still contain these insecure functions.

The fundamental problem of buffer overflows is that memory is "accidently" being overwritten that is interpreted as e.g. a function pointer or return address inside the program. The cause is that programmers often enough don't really care about input validation, including input length. Programmers still use insecure functions like `strcpy(3)`, `strcat(3)`, `sprintf(3)`, and so forth. These functions have the big disadvantage that they don't care about buffer sizes. Unless the programmers carefully checks buffer sizes before actually copying around strings and memory, it's very easy to produce programs with buffer overflows.

Basically, there are two types of buffer overflows that are usually exploited:

- Stack overflows
- Heap overflows

Stack overflows occur with buffers that are allocated on the stack, e.g. by declaring local variables or by allocating memory using `alloca(3)`. Heap overflows occur with buffers that are dynamically allocated on the heap, e.g. by calling functions like `malloc(3)` and friends.

It has been shown in many incidents that both stack and heap buffer overflows are exploitable, i.e. can be used to execute unauthorized and potentially malicious code. More information on stack and heap smashing can be found in [One96] and [Kae01].

1.2 The Ideas of ProPolice

ProPolice is a technology developed by Hiroaki Etoh and Kunikazu Yoda that tackles the problem of stack overflows. ProPolice is extensively described in [EY00]. It is developed as an extension to the well-known *gcc* (GNU C Compiler, part of the GNU Compiler Collection). According to the paper mentioned before, ProPolice is based on StackGuard.

The idea of ProPolice is quite simple: to protect the location of the arguments, the return address and the previous frame pointer of a function, a *guard variable* is introduced, which is initialized with a *guard value* at the function's entry point. When the functions exits, the guard variable is checked whether it still contains the guard value. If it doesn't, execution of the program is halted. The guard variable, also referred to as canary, is randomly generated.

1.3 ContraPolice

ContraPolice represents ProPolice's counterpart, as ContraPolice protects the dynamically allocated memory on the heap, while ProPolice protects the stack.

ContraPolice's concept is similar to ProPolice: when new memory is allocated (using `malloc(3)` and friends), ContraPolice places a *decoy* before and after the allocated memory (the decoy in ContraPolice jargon is what the canary is in ProPolice jargon).

Before leaving a library function handling buffers – no matter whether it's an insecure library function or not – the buffer's address that is currently handled is looked up whether it is registered in the list of dynamically allocated memory regions. In case such a memory region is found, a check function is called to check the decoys for correct values. If the values before and after the allocated memory don't match, an error message is printed and the execution of the program is halted. As with ProPolice, the decoy value is randomly generated.

2 Design

ContraPolice is an extension for the *libc*, which means it is bound to a certain libc implementation and (depending on the libc) to a platform. Currently, there's only the reference implementation for dietlibc (a small libc for Linux) available, but the concept itself can be implemented with other libcs, e.g. *glibc* or *OpenBSD libc*.

As explained before, when dynamically allocating memory, ContraPolice places decoys before and after the allocated memory. The decoy value is randomly generated. Real random numbers are preferred to pseudo-random numbers, since correctly guessing the decoy value would make it possible to overflow a buffer while bypassing ContraPolice.

ContraPolice keeps a list of all memory blocks that were dynamically allocated including their start address and their allocated size. The allocated size describes how much memory was actually requested. When a new memory block is requested, it is first allocated inside the `malloc(3)` routine, and then added to the list of memory blocks before returned to the program.

During program execution, any memory address can be checked by calling the library routine `cp_check()` (provided by ContraPolice) to check whether this memory address is inside a dynamically allocated memory block and if so, whether the decoy values of this memory block are still OK. In case they aren't, the program is immediately aborted. The algorithm of this check function is best described by the following pseudo code:

```
for i=1 to mem_blk.size do
  if mem_addr between mem_blk[i].start_addr and mem_blk[i].end_addr
  then
    check_decoy(mem_block[i])
  end
end
```

This means when a memory address is checked that is not dynamically allocated by calling `malloc(3)` and thus not in the list of memory blocks, it is simply ignored.

To enforce these checks, all functions that modify buffers in any way are modified to call `cp_check()` before returning. This means that if e.g. `strcpy(3)` copies a string of 50 characters into a buffer of 30 characters, and `cp_check()` is called before `strcpy(3)` returns, the program will abort, and a potential exploit that could have been planted into the program via this buffer overflow will never be executed.

When a dynamically allocated memory block is disposed by calling `free(3)`, the memory block is first checked using `cp_check()`, and then removed from the list of allocated memory blocks that ContraPolice keeps.

3 Implementation

3.1 dietlibc

The reference implementation of ContraPolice is based on dietlibc, a libc that is optimized for small size. I chose dietlibc over other libc implementations since the source code is very small and simple and easy to understand and modify. More information on dietlibc can be found in [Lei01].

3.2 Implementation of malloc(3)/free(3) in dietlibc

dietlibc's implementation of `malloc(3)`, `free(3)` and friends is very simple: since dietlibc is Linux only, it uses a certain property of Linux's `mmap(2)` system call: when a file descriptor of -1 is passed, anonymous memory is allocated, which then can be deallocated by calling `munmap(2)`.

Since `munmap(2)` requires additional information that is not passed in to the `free(3)` call, the memory management routines have to keep track of this information by themselves. This is done by allocating more memory than requested. This newly allocated memory first contains this information, which is kept in an own structure `__alloc_t`. The memory right after this structure is returned by `malloc(3)`. When deallocating the memory again, `free(3)` only has to take the pointer to the memory block to deallocate and has to take a look before its beginning. This is exactly where the `__alloc_t` structure resides. For convinience, dietlibc internally contains two preprocessor macros that make it easy to compute the memory address to return from the memory address returned by `mmap(2)` and vice versa.

```
#define BLOCK_START(b)   (((void*)(b))-sizeof(__alloc_t))
#define BLOCK_RET(b)     (((void*)(b))+sizeof(__alloc_t))
```

Since the granularity of the `mmap(2)` system call is actually huge (e.g. 4 kB on i386), dietlibc realizes a simple list-based memory management for small memory blocks in user space. This raises the code size of the `malloc(3)` and `free(3)` library functions a little bit, but significantly lowers the memory footprint especially for programs that dynamically allocate a great number of relatively small memory blocks.

3.3 Extensions done for ContraPolice

To keep track of all allocated memory blocks, ContraPolice needs to keep a list of them. In the reference implementation, this is a simple linked list. New memory blocks are added to the head of the list (thus making it $O(1)$ complex and keeping `malloc(3)` efficient). When a memory block is deallocated, it has to be removed from the list. This means that the whole list has to be iterated over. This makes the deallocation routine more complex than the allocation routine, i.e. $O(n)$.

To be able to strictly check for possible heap overflows, ContraPolice needs to keep track of how much memory was exactly allocated. What is also required is the decoy value, which is also stored in the `__alloc_t`. The other decoy value is stored in a `__decoy_t` structure. The `__alloc_t` structure itself looks like this:

```
typedef struct __alloc {
  void*  next;
  size_t size;
  struct __alloc * cp_next;
  size_t alloc_size;
  uint32_t decoy;
} __alloc_t;
```

To briefly explain the elements of this structure:

- `next`: a pointer needed for dietlibc's list-based memory management.
- `size`: contains the actual size of the memory block allocated via `mmap(2)`.
- `cp_next`: a pointer that points to the next element in the list of memory blocks.
- `alloc_size`: the memory block size that was requested via `malloc(3)`, i.e. the number of bytes between the __alloc_t structure and the __decoy_t structure.

The __decoy_t structure is as simple as that:

```
typedef struct {
  uint32_t decoy;
} __decoy_t;
```

As we've learned before, general information about the allocated memory is placed the actual memory block that is returned by `malloc(3)`. What we still need is the second decoy value. This second decoy value is placed after the returned memory block.

Whenever `cp_check()` is called, the list of allocated memory block is iterated over, to check whether a certain address is dynamically allocated, and if so, whether the corresponding memory block's decoy values are still intact.

The decoy value itself is generated by reading from `/dev/urandom`, which is generally a good place provided by Linux to get good pseudo-random numbers. In case that not enough random data can be read from this device, ContraPolice switches back to pseudo-random numbers provided by the standard library. The quality of the random numbers should be as high as possible, to make it as difficult as possible to guess the right value for potential attackers.

4 Conclusions

In this paper I gave an introduction into a concept of a heap-smashing protection built into a libc, including an explanation of a reference implementation of ContraPolice for dietlibc.

One major of this implementation of ContraPolice is that every time a linked list has to be iterated over, which generally makes programs and libraries using ContraPolice noticeably slower, especially with a lot of allocated memory blocks. The price for this performance penalty is an improved protection against heap buffer overflows, even for programs that are written insecurely.

In the future, the ContraPolice concepts will be implemented for other libcs. I hope that it will make it into the libcs for mainstream free Unix-like operating systems, improving everybody's daily life computer security.

References

[EY00] ETOH, Hiroaki ; YODA, Kunikazu. *Protecting from stack-smashing attacks*. `http://www.research.ibm.com/trl/projects/security/ssp/main.html`. 2000

[Kae01] KAEMPF, Michel. *Vudo - An object superstitiously believed to embody magical powers*. `http://www.phrack.org/phrack/57/p57-0x08`. 2001

[Lei01] VON LEITNER, Felix. *diet libc*. `http://www.fefe.de/dietlibc/talk.pdf`. 2001

[One96] ONE, Aleph. *Smashing The Stack For Fun And Profit*. `http://www.phrack.org/phrack/49/P49-14`. 1996