Hunting birds

Yuki Koike

NADA Junior and Senior High School
8-5-1, Uozaki-Kitamachi, Higashinada-ku, Kobe-shi, Hyogo 658-0082, JAPAN
poteticalbee@gmail.com

Abstract Stack Smashing Protection (SSP) is one of the oldest and fundamental protections against memory corruption exploits. Employing stack canaries to detect malicious corruption, SSP has proved to make exploiting memory corruption bugs to be more difficult. Stack canaries verify if a buffer has been overflown by checking the integrity of a value stored immediately after the buffer. Although some papers presented some trivial methods in Linux to defeat stack canaries, they require certain function pointers to be overwritten or only work in certain circumstances. Therefore, they are not so useful as a versatile exploit technique in the present because they require rare configuration and environments that are unlikely to be found in the real world. This paper proposes a new technique called "Master Canary Forging", which provides practical use in terms of adaptability and feasibility for killing stack canaries with no need for using other exploitation methods which modify other function pointers such as GOT Overwrites or SEH Overwrites.

1 Introduction

Stack Smashing Protection (SSP) is a security exploit mitigation that compilers use to detect during runtime whether a stack frame has been overwritten and to terminate the process if it has in order to mitigate the possibility of arbitrary execution of code by an attacker. The idea and first implementation of SSP was released in 1998 and has been modified and improved on ever since. A stack based buffer overflow is a vulnerability that allows an attacker to write passed the intended memory space allocated for a variable into the area on a call stack. For programs compiled with SSP, the attacker would have to overwrite pass the canary before reaching the frame pointer and return address on the call stack. It could be said that this is the most straightforward and easiest way to exploit memory corruption bugs since it is possible to directly tamper with call stacks, which are responsible for the program flow. This is the motivation for creating a method of bypassing these canary checks.

A stack canary can either be a random value hard to predict value that should not be able to be guessed or brute-forced, or a terminator value that has been carefully selected to prevent overwriting any data passed it. If the random value would be able to be predicted or if the attacker could write pass the terminator value, then the attacker would be able to successfully overwrite the call stack and take control of the program. Therefore, it is necessary make sure

that the random value is truly unpredictable and that the terminator value is unable to be written passed. Currently, stack canaries can be separated into three major groups: a) terminator, b) random, and c) random XOR. Most modern Linux distributions use random canaries. In the current glibc implementation, *_dl_setup_stack_chk_guard* in ld.so stores random values generated from random devices such as */dev/random* or */dev/urandom* in the fixed locations of the TLS segments or the BSS segments when a program is running. These values are called master canaries. Programs are compiled to be able to read these master canaries from the designated places in accordance with the design of glibc. Thus, different random values are generated every time the program is launched. These values are taken from the master canaries and placed after the allocated program variables and before the base pointer and return address when a function is called. When the function ends, the canary value is then checked to make sure that it matches with the master canary to find out that it has not been changed.

There are mainly two methods to bypass stack canaries with stack based buffer overflows: a) completely avoid the canary validation by abusing other function pointers and b) bypassing the canary validation by somehow overwriting it with the correct value. The techniques covered in [1], [2], and [3] give examples of a) and b). Master Canary Forging, described in this paper, does not deal with the first type and, as the name suggests, corrupts the master canaries. This is opposed to the techniques explained in [1] and [2], that take the opposite approach by finding out the values of the master canaries and overwriting the canary values to match them. This attack would be much more difficult if random XOR canaries were used, as they are in Windows, as a vulnerability that leaks stack address would also be necessary. The original idea for Master Canary Forging is mentioned in [4], however, there are only works on non-ASLR systems. I have improved on this idea to make Master Canary Forging work even if ASLR enabled. This is obvious when the target architecture does not support reading from the TLS segment because the BSS segment is always mapped at fixed addresses. Hence, this paper covers only the x86_64 architecture which uses the TLS segments for reading canaries.

2 Environment

This paper assumes the following environments:

- Linux Kernel 3.19
- Glibc 2.21
- GCC 4.9.2
- x86_64

Note that Master Canary Forging runs regardless whether or not DEP, ASLR, PIE, and RELRO are enabled.

3 Attack Outline

Master Canary Forging consists of the following phases:

1. Establish Mapping: Create a mapping by invoking *mmap* to make it and the TLS segment successive.
2. Master Canary Overwrite: Overflow a buffer in the mapping in order to forge a master canary.

3. Stack Canary Overwrite: Overflow a stack based buffer overflow.

Therefore, Master Canary Forging requires three conditions: the capability of calling *mmap* with specific arguments, the capability of overflowing a buffer in the mapping, and the capability of causing a stack based buffer overflow.

4 Establish Mapping

- The *addr* argument is NULL.
- *MAP_FIXED* and *MAP_32BIT* are not specified by the *flags* parameter.
- *–addr-compat-layout* is not indicated by the *setarch* command.
- The value of *RLIMIT_STACK* is not *RLIM_INFINITY*.
- */proc/sys/vm/legacy_va_layout* is set as 0.

In the *mmap* implementation of Linux Kernel 3.19 x86_64, if these conditions are satisfied, then mapping addresses are computed as follows in *unmapped_area_topdown*.

```
    /* Check highest gap, which does not precede any
       rbtree node */
    gap_start = mm->highest_vm_end;
    if (gap_start <= high_limit)
        goto found_highest;

    ...

    while (true) {
        /* Visit right subtree if it looks promising */
        gap_start = vma->vm_prev ? vma->
            vm_prev->vm_end : 0;
        if (gap_start <= high_limit && vma->
            vm_rb.rb_right) {
            struct vm_area_struct *right =
            rb_entry(vma->vm_rb.rb_right,
            struct vm_area_struct, vm_rb);
            if (right->rb_subtree_gap >= length) {
                vma = right;
                continue;
            }
        }

check_current:
        /* Check if current node has a suitable gap
           */
        gap_end = vma->vm_start;
        if (gap_end < low_limit)
            return -ENOMEM;
        if (gap_start <= high_limit && gap_end -
            gap_start >= length)
            goto found;

        /* Visit left subtree if it looks promising */
        if (vma->vm_rb.rb_left) {
            struct vm_area_struct *left =
                rb_entry(vma->vm_rb.rb_left,
                    struct vm_area_struct,
                    vm_rb);
            if (left->rb_subtree_gap >= length)
                {
                vma = left;
                continue;
            }
        }

        /* Go back up the rbtree to find next
           candidate node */
        while (true) {
            struct rb_node *prev = &vma->vm_rb;
            if (!rb_parent(prev))
                return -ENOMEM;
            vma = rb_entry(rb_parent(prev),
            struct vm_area_struct, vm_rb);
            if (prev == vma->vm_rb.rb_right) {
                gap_start = vma->vm_prev ? vma
                    ->vm_prev->vm_end : 0;
                goto check_current;
            }
        }
    }

found:
```

```
    /* We found a suitable gap. Clip it with the
         original high_limit. */
    if (gap_end > info->high_limit)
        gap_end = info->high_limit;

found_highest:
    /* Compute highest gap address at the desired
         alignment */
    gap_end -= info->length;
    gap_end -= (gap_end - info->align_offset) &
         info->align_mask;

    return gap_end;
```

Looking at this, in the above case, *mmap* adopts the topdown method which establishes a mapping at the highest address in free space capable of being mapped. On the contrary, it selects the bottomup method if those conditions not satisfied. In either case, a new mapped region is always adjacent to some region which has already been mapped unless the highest region is unmapped. That means that it is possible to overwrite all of the master canary values if the mapped area for a buffer overflow is located in front of the TLS segment, and all areas between them are consecutive and writable. Although these requirements may at first seem difficult to be met, they are actually fulfilled in most applications. This is due to the front of the TLS segment being empty at all times unless the application makes its own call of *mmap* because the TLS segment is the second to last to be mapped by *_dl_allocate_tls_storage*, and meanwhile there is no hole created by munmap between any regions, and the heap region, which is the last, is allocated by another system call, *sbrk*. In this scenario, the attacker must then consider how to directly call *mmap*. *mmap* is essentially a pretty low level system call, so few applications but some exceptions using *mmap* in their code. However, there is a widely used glibc function which calls *mmap*: *malloc*. *malloc* is well known to deal with allocated requests that exceed *MMAP_THRESHOLD* bytes by creating a new pool with *mmap* [5]. This trait enables an attacker to make a heap area continuing to the TLS segment if it is possible to send specific sized allocation requests. Summarizing the above, the conditions mentioned in chapter 3 can be rephrased in the following:

- The capability to control allocation sizes of *malloc*
- A heap based buffer overflow
- A stack based buffer overflow

5 Proof of Concept

Code 1 poc.c

```
/*
 * gcc poc.c -fstack-protector-all -Wl,-z,now,-z
      ,-relro
 */

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

void stack_overflow(void) {
    char stack_buf[16];

    fread(stack_buf, 1, 48, stdin);
    return;
}

int main(void) {
    size_t alloc_size = 0;
    size_t read_size = 0;
    char *heap_buf;

    if (scanf("%zu", &alloc_size) != 1) return -1;
    if (scanf("%zu", &read_size) != 1) return -1;
```

```
    heap_buf = (char*)malloc(alloc_size);
    if (!heap_buf) return -1;

    fread(heap_buf, sizeof(char), read_size, stdin);
    stack_overflow();
    free(heap_buf);
    return 0;
}
```

Code 2 exploit.py

```
print "{}".format(0x21000)
print "{}".format(0x23720)
print "a" * (0x23720+0x30)
```

The binary is available at: https://github.com/potetisensei/MasterCanaryForging-PoC/.

6 Prevention

I recommend using random XOR canaries in order to prevent this exploitation method and to increase the sources of entropy enough to ensure that they are truly unpredictable.

7 Conclusion

This paper describes a new exploitation technique to defeat stack canaries called Master Canary Forging, which uses *malloc* to utilize the properties of *mmap* to establish successive memory mappings in order to overwrite the master canary values. It is architecture dependent and currently has been proven to work on the x86_64 architecture with the latest stable Linux Kernel, glibc, and gcc. There are some other cases where Master Canary Forging will succeed on other architectures but is out of scope of this paper.

8 Acknowledgments

I thank Isaac Mathis for his kind and swift feedback and proofreading, and Yuma Kurogome who double-checked my PoC.

References

[1] Paul Rascagneres. Stack Smashing Protector http://www.hackitoergosum.org/2010/HES2010-prascagneres-Stack-Smashing-Protector-in-FreeBSD.pdf

[2] Ben Hawkes. Exploiting OpenBSD http://inertiawar.com/openbsd/hawkes_openbsd.pdf

[3] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/defeat-stackguard.pdf

[4] Hagen Fritsch. Stack Smashing as of Today https://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf

[5] malloc(3) - Linux man page http://linux.die.net/man/3/malloc