Stack Canaries

G. Lettieri

17 October 2023

1 Introduction

We will now introduce one of the first "mitigations" that have been invented to counter attacks that exploit software bugs. The idea behind the mitigation strategy is that, no matter how hard we try to write correct programs, there will always be unknown bugs in them. A mitigation attempts to limit the damage caused by these unknown bugs, in cases where they are discovered and exploited by the attackers before they are fixed.

Stack canaries are a mitigation that targets stack-based buffer overflow attacks. It works by exploiting one of the limitations of this type of attacks, which is that the attacker must overwrite *all* the bytes between the overflowed buffer and the control data (i.e., the saved registers and the return address). The idea is to place a value—the canary—between the local variables and the control data of each function stack frame. Thus, the attacker must overwrite the canary before she can overwrite the control data. If overwriting the canary is either impossible or detectable, the attack is blocked.

The name "canary" refers to the actual canaries used in coal mines at the beginning of the 20th century. The miners would bring a cage with a canary in it to detect toxic gas leaks. Because canaries are small and breathe much more oxygen than other small animals, they will feel the effects of the gas long before the miners, giving them time to leave the mine. Similarly, functions can detect that attacker bytes (the poisonous gas) have leaked from a buffer by checking the "health" of the stack canary before trusting the control data.

This idea translates into a change of the prologue and epilogue of each function, and as such it's naturally implemented in the compiler. The canary-enabled prologue must push the canary on the stack, immediately after the control data (return address and saved registers) and before the allocation of the local variables. The canary-enabled epilogue must check that the canary still contains the original value, before restoring the saved registers and returning. If the canary has been changed, a possible attack has been detected and the process must be aborted. This turns a bug that could lead to a compromise of the process's credentials into a more benign denial of service.

2 Kinds of canaries

Three types of canaries have been proposed, each one with its own strenghts and weaknesses:

Random: the canary is a random number, unknown to the attacker;

Terminator: the canary contains characters that stop most string functions (newline, null byte, linefeed, -1);

XOR: the canary is the XOR of a random value and the saved return address.

Random canaries are good if the attacker cannot guess them. Unfortunately, memory leak bugs can reveal the canary's value, rendering them useless. Terminator canaries, on the other hand, are constant and well known to the attacker. They block the attacks in a different way. For example, consider a `gets()`-based overflow like the one we have studied in the previous lecture. We know that the attacker cannot have a newline in her payload, because otherwise the `gets()` will not copy all the bytes that come after it. But if the canary contains a newline and the attacker overwrites it with something else, it will be detected. Unfortunately, there are bugs that involve `memcpy()`, `read()`, or even custom hand-made code, and these bugs are either not blocked by any particular byte value, or by characters that are not in the canary.

Both types of canaries are useless against bugs that allow the attackers to overwrite arbitrary memory addresses, since these types of attacks can overwrite the saved return address without touching the canary. The XOR canaries attempt to block these attacks by merging the original stored rip address with the random canary. However, a memory leak can reveal both the the XOR canary and the stored rip address, thus exposing the random value and so on. Also, an arbitrary-memory-write bug may allow the attacker to overwrite something else in the control data (e.g., the saved **`rbp`**) without touching either the XOR canary or the return address. This can be countered by XORing even more control data with the canary, but the attacker may find other useful things to overwrite (e.g., function pointers elsewhere in memory). Combinations of two or more types of canaries are possible, but the they have other problems. For example, a canary containing both terminator and random characters has reduced entropy compared to a completely random canary, and may therefore be easier to guess.

In summary, canaries can be, and often have been, defeated by memory leaks and/or attacks that do not rely on stack-based buffer overflows (see, for example, the format string vulnerabilities in another lecture). This is not surprising, since the canaries were specifically designed to exploit a peculiarity of buffer overflows, especially stack-based ones. To these "genetic" weaknesses, however, we must add other weaknesses that may arise from their implementation in a particular system. Indeed, in real systems, the implementation may be particularly weak due to backward compatibility constrains or performance-versus-security considerations.

So why use them at all? Like all mitigations, they do not block all possible attacks. However, the idea is that by adding enough mitigations, each one targeting a specific type of bug, we can block most, if not all, existing exploitable attacks.

There is a very important mental trap that we should avoid: thinking that an attack is blocked just because it is "difficult to exploit", and that therefore we don't need to worry about bugs. The perception of difficulty only comes from our inexperience as attackers, and this is why we try to learn attackers' known techniques: to have a better idea of what is really difficult and what is just a nuisance. Bugs need to be found and fixed. Mitigations are just a last line of defense.

3 Implementation in GNU/Linux

Linux systems that use the GNU C library and `gcc` (i.e., most of the Linux systems) implement stack canaries as a collaboration between the kernel, the compiler and the C library. The workflow is as follows:

1. During each `execve()`, the kernel places a random value in the stack of the new virtual memory;
2. The C runtime initialization functions that come with the GNU libc use this value to compute the canary and place it in a well known location in the process's memory;
3. the function prologue generated by `gcc` takes this global canary and pushes it on the stack; the function epilogue checks if the local canary matches the global one, and aborts the process if they differ.

We can already make some considerations. The canary is always the same for the entire lifetime of a process. There is a new canary only when `execve()` is called: a `fork()`ed process will use the same canary as its parent. The canary resides in many places in memory: there always is the global copy and, at any given moment, there is a copy in the stack frames of all currently active functions. Other copies, coming from previously returned functions, may be found in uninitialized variables of currently active functions. The attacker only needs a memory leak bug, in any part of the program, that she can use to read any of these copies. Then she can use a buffer overflow bug in some other part of the program to successfully overwrite the canary and the control data it was supposed to protect. Note that the attacker now has to find two exploitable bugs instead of just one, so the mitigation reduces the attacker's chances somewhat.

Let us now examine the implementation in more detail.

3.1 The kernel

Upon completion of the `execve()` system call, the kernel puts 16 random byes on the stack, typically just above the argument and environment strings. It then puts a pointer to these bytes into an entry of the *auxiliary vector*. This is a data structure that the kernel pushes onto the process stack, just below the environment array. This data structure contains various information about the process and the program and is mainly used by the dynamic loader, which we will study in another lecture. The auxiliary vector is a vector structures. Each structure occupies two stack lines: The first line contains a numeric "tag" that identifies the type of information contained into the second line. The `AT_RANDOM` tag (value 25, hex 19) is the one we are interested in: the second line of the entry with this tag contains the pointer to the random bytes.

3.2 The GNU C library

The GNU C library contains some object files that are linked with all programs by default. They contain the C runtime initialization and cleanup routines. The `_start` entry point of our program is defined in one of these files. It contains a small assembly program that calls the `__libc_start_main()` function, which is still part of the C library and is written in C. This function performs many initializations and then calls our `main()`. Among these initializations there is the creation of the canary value, which is very simple: the canary coincides with the least significant bytes of the random number generated by the kernel (4 bytes on 32b systems and 8 bytes on 64b systems). However, the least significant byte is replaced by 0. Therefore, the canary is a mixture of a "terminator" and a "random" canary. Note that for 32b this means that only three bytes of the canary are random, which makes it not difficult to guess by brute-forcing. We can also see that there is still another place where the canary can be (essentially) read: the random value stored by the kernel on the stack.

The canary is stored in the *Thread Control Block* (TCB). This is a per-thread data structure used by standard libraries to hold information that depends on the current thread. For the entire lifetime of a thread, a pointer to this structure is contained in the (hidden part of) the **`fs`** register. This is one of the two *segment selector* registers (the other is **`gs`**) that are still in use on the AMD64 architecture. They contain a constant virtual address that can be added to any memory operand address. In assembly syntax you can use it by prepending "**`fs:`**" to a memory operand. The **`fs`** register is initialized by `__libc_start_main()`[1].

3.3 The gcc compiler

The `gcc` compiler will add canary support to the compiled program if the `stack-protector` option is enabled. In current Linux distributions, this is enabled by default and can be disabled by adding the `-fno-stack-protector` option to the `gcc` command line.

When canaries are enabled, the prologue of canary-protected functions becomes:

[1]The function uses the Linux-specific `arch_prctl()` system call to initialize the register, since it cannot be written from userspace.

```
 1    ; save the old frame pointer
 2    push rbp
 3    ; (maybe push other registers)
 4    ; create the new frame pointer
 5    mov  rbp, rsp
 6    ; reserve space for the local variables + the canary
 7    sub  rsp, x
 8    ; copy the global canary in the current frame
 9    mov rax, QWORD PTR fs:0x28
10    mov QWORD PTR [rbp-8], rax
```

Lines 1–7 contain a standard prologue, except for the need to reserve space for the canary in addition to the local variables. Lines 9 and 10 are new: line 9 reads the global canary from offset `0x28` in the TCB and line 10 copies the canary just above the saved frame pointer. Note that, if the compiler has to save other registers besides the old frame pointer (see the comment at line 3), the canary will be stored above them.

The canary protected epilogue is:

```
 1    ; compare the global canary with the local copy
 2    mov rax, QWORD PTR [rbp-8]
 3    sub rax, QWORD PTR fs:0x28
 4    je good
 5    ; abort if the local canary has been modified
 6    call __stack_chk_fail@plt ; doesn't return
 7  good:
 8    ; restore old rsp and rbp
 9    leave
10    ; return to the caller
11    ret
```

Lines 1–7 are new, while the others are nothing more than the standard epilogue.

The new instructions add a bit of overhead to the function, so `gcc` only adds them where it thinks they are really needed. Basically, only in functions that declare sufficiently large array variables [2]

The `__stack_chk_fail` function prints an error message on standard error and aborts the process. For the time being, ignore the strange `@plt` suffix in the function name: It is a reference to the linker-generated Procedure Linkage Table (PLT). We will study it in another lecture, where we will learn how it introduces yet another opportunity for attackers.

[2]Search for "-fstack-protector" in the `gcc` manpage for full details.