

# Secure Coding in C and C++

## Exercise #4: Using Valgrind to Find Memory Errors

Software Engineering Institute  
Carnegie Mellon University  
Pittsburgh, PA 15213



Software Engineering Institute

Carnegie Mellon University

© 2015 Carnegie Mellon University

# Notices

---

Copyright 2022 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material was prepared for the exclusive use of Trainees of online & offline course and may not be used for any other purpose without the written consent of [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

This material is distributed by the Software Engineering Institute (SEI) only to course attendees for their own individual study.

Except for any U.S. government purposes described herein, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at [permission@sei.cmu.edu](mailto:permission@sei.cmu.edu).

Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.

This material and exercises include and/or can make use of certain third party software ("Third Party Software"), which is subject to its own license. The Third Party Software that is used by this material and exercises are dependent upon your system configuration, but typically includes the software identified in the documentation and/or ReadMe files. By using this material and exercises, You agree to comply with any and all relevant Third Party Software terms and conditions contained in any such Third Party Software or separate license file distributed with such Third Party Software. The parties who own the Third Party Software ("Third Party Licensors") are intended third party beneficiaries to this License with respect to the terms applicable to their Third Party Software. Third Party Software licenses only apply to the Third Party Software and not any other portion of the material as a whole.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

CERT® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM22-0263

# Sample Program

---

## Caesar cipher decryption program

- Implements simple rotation cipher
- Takes input from files

In a real usage scenario, the decrypted file and the keys file must be kept secret from unauthorized users.

- Should only be usable by intended user
- The secret file can be used by anyone; it's protected by encryption!

# Usage

---

The program accepts a command line argument:

**Usage:** %s **secret\_file** **keys\_file** [**output\_file**]

The **secret\_file** argument specifies the name of the file containing the encrypted text.

The **keys\_file** argument specifies the name of the file containing the corresponding “keys” to decrypt each line of the encrypted text.

- **keys\_file** must live in home directory, or a subdirectory.
- **keys\_file** can only be read with **root** privileges

The program also accepts an optional **output\_file** argument.

- If **output\_file** is not specified, the program prints the output to **stdout**.
- Otherwise **output\_file** must be placed in home directory, or a subdirectory.

# The Input Files

---

All of the files involved are just character files.

Each line contains the ciphertext and corresponding “key” (number of chars to rotate). For example:

Ciphertext	Key	Plaintext
Lzak ak s lwk1	8	This is a test

The lines are delimited by EOL.

A working set of example files is included.

# Using Valgrind

---

Compile application with debugging symbols and low optimization if possible (**-g3** and **-O0**)

- allows Valgrind to produce more informative output, just as with a debugger

Invocation:

```
$ valgrind --leak-check=full myprog \  
arg1 arg2...
```

The **--leak-check=full** option enables memory leak checking

# Exercise

---

Review and repair the code.

- compile and test with the supplied valid input files
- quick manual code reading and identification of potential problem areas
- construct various invalid inputs, rerun under Valgrind using those inputs, and record the errors produced

Use reference material.

- Valgrind documentation ([/usr/share/doc/valgrind](#), man page)
- C standard
- man / help pages
- CERT Secure Coding standards
- *Secure Coding in C and C++*

# Exercise

---

Find memory errors  
using Valgrind  
(30 minutes)





# Double-Free in `usage()` 1

What if we supply no arguments?

```
$ valgrind ./caesar
```

```
[...]
```

```
sorry, user
```

```
Usage: caesar secret_file keys_file [output_file]
```

```
==19557== Invalid free() / delete / delete[]
```

```
==19557==    at 0x401D240: free (vg_replace_malloc.c:233)
```

```
==19557==    by 0x8048706: main (caesar.c:27)
```

```
==19557== Address 0x4160028 is 0 bytes inside a block of size 80 free'd
```

```
==19557==    at 0x401D240: free (vg_replace_malloc.c:233)
```

```
==19557==    by 0x8048A3D: usage (caesar.c:78)
```

```
==19557==    by 0x80486FB: main (caesar.c:26)
```

```
==19557==
```

```
==19557== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 11 from 1)
```

# Double-Free in `usage()` 2

---

Once in `usage()`:

```
fprintf(stderr, errmsg);  
free(errmsg);
```

And once again in `main()`:

```
usage(errmsg);  
free(errmsg);
```

**Note:** Modern versions of the GNU libc will detect this as well:

```
*** glibc detected *** double free or corruption (top): 0x0804a008 ***  
Aborted
```

# Long Username Buffer Overflow

What if we supply a long username?

```
$ USER=`perl -e 'print "A" x 256;'` valgrind ./caesar
```

```
[...lots of invalid read/write messages...]
```

```
==19577== Invalid read of size 1
```

```
==19577==      at 0x406B0E7: vfprintf (in /lib/tls/i686/cmov/libc-2.3.6.so)
```

```
==19577==      by 0x406AD7B: buffered_vfprintf (in /lib/tls/i686/cmov/libc-...
```

```
==19577==      by 0x406AFBA: vfprintf (in /lib/tls/i686/cmov/libc-2.3.6.so)
```

```
==19577==      by 0x40736AE: fprintf (in /lib/tls/i686/cmov/libc-2.3.6.so)
```

```
==19577==      by 0x8048A32: usage (caesar.c:77)
```

```
==19577==      by 0x80486FB: main (caesar.c:26)
```

```
==19577== Address 0x4160162 is not stack'd, malloc'd or (recently) free'd  
sorry, [AA...]
```

```
Usage: caesar secret_file keys_file [output_file]
```

```
[...]
```

```
==19577== ERROR SUMMARY: 940 errors from 9 contexts (suppressed: 11 from 1)
```

# The `getenv()` Function

---

The contents of the `USER` environment variable are supplied to the usage error message without any bounds checking:

```
errmsg = (char *)malloc(LINELENGTH);  
sprintf(errmsg, "...", getenv("USER"));
```



No length check

# Input Line Buffer Overflows <sub>1</sub>

---

Spot the problem:

```
#define LINELENGTH 80
[...]  
if (! (inbuf = malloc(LINELENGTH)))  
    errx(1, "Couldn't allocate memory.");  
while (fgets(inbuf, 100, infile) ...
```

# Input Line Buffer Overflows 2

What if we supply an encrypted file with long lines?

```
$ valgrind ./caesar bad_encrypted.txt keys.txt
[...lots of invalid read/write messages...]
==22501== Invalid write of size 1
==22501==    at 0x401EB42: memcpy (mc_replace_strmem.c:406)
==22501==    by 0x4085102: _IO_getline_info (in /lib/tls/...
==22501==    by 0x4084FEE: _IO_getline (in /lib/tls/...
==22501==    by 0x4083F18: fgets (in /lib/tls/i686/cmov/libc-2.3.6.so)
==22501==    by 0x804888D: main (caesar.c:46)
==22501== Address 0x41603A7 is 15 bytes after a block of size 80 alloc'd
==22501==    at 0x401C621: malloc (vg_replace_malloc.c:149)
==22501==    by 0x80487CA: main (caesar.c:43)
[...]
==22501== ERROR SUMMARY: 52 errors from 7 contexts (suppressed: 11 from 1)
==22501== malloc/free: in use at exit: 2,032 bytes in 27 blocks.
==22501== malloc/free: 27 allocs, 0 frees, 2,032 bytes allocated.
```

# Uninitialized Variable

What happens when an invalid key is supplied?

```
$ valgrind ./caesar encrypted.txt bad_keys.txt
```

```
[...]
```

```
caesar: bad rotation value
```

```
==20338== Syscall param exit_group(exit_code) contains uninitialised byte(s)
```

```
==20338==    at 0x40BC4F4: _Exit (in /lib/tls/i686/cmov/libc-2.3.6.so)
```

```
==20338==    by 0x40F8092: errx (in /lib/tls/i686/cmov/libc-2.3.6.so)
```

```
==20338==    by 0x80488CC: decrypt (caesar.c:62)
```

```
==20338==    by 0x8048848: main (caesar.c:51)
```

```
==20338==
```

```
==20338== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 11 from 1)
```

In `decrypt()`:

```
int i;
```

```
[...]
```

```
if ((rot < 0) || (rot >= 26))
```

```
    errx(i, "bad rotation value");
```

Perhaps the programmer meant 1 instead of i?

# Valgrind Diagnostic

---

```
==4928== Conditional jump or move depends on uninitialized value(s)
==4928==    at 0x40239E7: strlen (mc_replace_strmem.c:242)
==4928==    by 0x4089127: fputs (in /lib/tls/i686/cmov/libc-2.7.so)
==4928==    by 0x804888F: main (caesar.c:52)
```

```
outbuf = decrypt(inbuf, atoi(keystr));
fputs(outbuf, (oflag ? outfile : stdout));
```

**outbuf is initialized, but  
not null-terminated!**



# Null-termination

---

```
if (! (outbuf = malloc (LINELENGTH) ) )  
    err (1, NULL) ;  
i = 0 ;  
while (ch = msg[i]) {  
    outbuf[i] = /* . . . */ ch ;  
    ++i ;  
}  
outbuf[i] = '\\0' ;  
return outbuf ;
```

Failed to null-terminate  
the NTBS outbuf

# Memory Leaks

---

There are also lots of memory leaks in this code:

```
$ valgrind --leak-check=full ./caesar bad_encrypted.txt keys.txt
[...]  
==6436== 1,300 bytes in 13 blocks are definitely lost in loss record 4 of 4  
==6436==    at 0x4022AB8: malloc (vg_replace_malloc.c:207)  
==6436==    by 0x80488FB: decrypt (caesar.c:64)  
==6436==    by 0x8048863: main (caesar.c:51)  
==6436==  
==6436== LEAK SUMMARY:  
==6436==    definitely lost: 1,432 bytes in 27 blocks.  
==6436==    possibly lost: 0 bytes in 0 blocks.  
==6436==    still reachable: 704 bytes in 2 blocks.  
==6436==    suppressed: 0 bytes in 0 blocks.
```

Leaks such as these can allow an attacker to cause a denial of service on an affected program.

# Bonus: Format String Vulnerability

In `usage()`:

Recall that `errmsg` is constructed from `getenv("USER")`

```
fprintf(stderr, errmsg);
```

Missing format specifier

Valgrind doesn't really detect this error but provides some helpful output over `gdb`:

```
$ USER="user%n%n%n%n" valgrind ./caesar
==19584== Process terminating with default action of signal 11 (SIGSEGV)
==19584== Bad permissions for mapped region at address 0x80486FC
[...]
==19601== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
```

# Questions

