Concurrency issues

PA193 – Secure coding

Petr Švenda
Zdeněk Říha
Faculty of Informatics, Masaryk University, Brno, CZ

CROCS
Centre for Research on Cryptography and Security

Introduction - Terminology

- Program
  - File on disk
- Process
  - Executed program
- Thread
  - Entity within a process
  - A single sequence of instructions
  - CPU is scheduled to threads
- Memory, files descriptors are allocated to processes

Concurrency

- Parallel or interleaved (time multiplex) running of processes and threads can be a source of problems
  - single CPU
  - HTT
  - Multiples CPUs (SMP, AMP)
- Code in processes/threads is not executed atomically

IPC

- OS isolates processes
- To interact processes have to use IPC offered by the operating system
- IPC = Inter Process Communication

IPC

- Semaphor
- Mutex/futex
- Pipe (unnamed, named)
- Shared memory
- Messages
- Signals
- RPC, OLE, DDE, Clipboard

Other resources

- Files
  - filesystem is a shared resource used by many processes, and some procesess may interfere with its use by other processes…

Consistency problem

- The tasks:
  - 3 variables: A, B, C
  - 2 threads: T1, T2
  - Thread T1 computes C = A+B
  - Thread T2 moves the value X from A to B
- Expected behaviour
  - T2 does A = A-X and B = B+X
  - T1 computes a constant C, i.e. A + B does not change
- What if:
  - T1 calculates A+B
  - after T2 has done A = A-X
  - but before T2 does B = B+X
  - then T1 does not get correct result C = A+B

Consistency problem

- Similarly if two threads use one shared variable *x*
  - T1 is running x++
  - T2 is running x—
- But x++ is implemented as
  - **register1 = x**
  - **register1 = register1 + 1**
  - **x = register1**

Consistency problem

- Let's start with x=5 and run T1, T2 concurrently
  - T1: **register1 = x** (*register1 = 5*)
  - T1: **register1 = register1 + 1** (*register1 = 6*)
  - T2: **register2 = counter** (*register2 = 5*)
  - T2 : **register2 = register2 – 1** (*register2 = 4*)
  - T1 : **counter = register1** (*counter = 6*)
  - T2 : **counter = register2** (*counter = 4*)
- The result is 4 instead of 5
  - Inconsistency!
- Memory
  - Shared memory between processes as IPC
  - Memory shared between threads of a single process

Race condition - definition

A race condition or race hazard is the behavior of an electronic or software system where the output is dependent on the **sequence or timing** of other uncontrollable events. It becomes a bug when events don't happen in the order that the programmer intended.

The term originates with the idea of two signals racing each other to influence the output first.

Race conditions can occur in electronics systems, especially logic circuits, and in computer software, especially multithreaded or distributed programs.

Source: Wikipedia

Race condition - definition

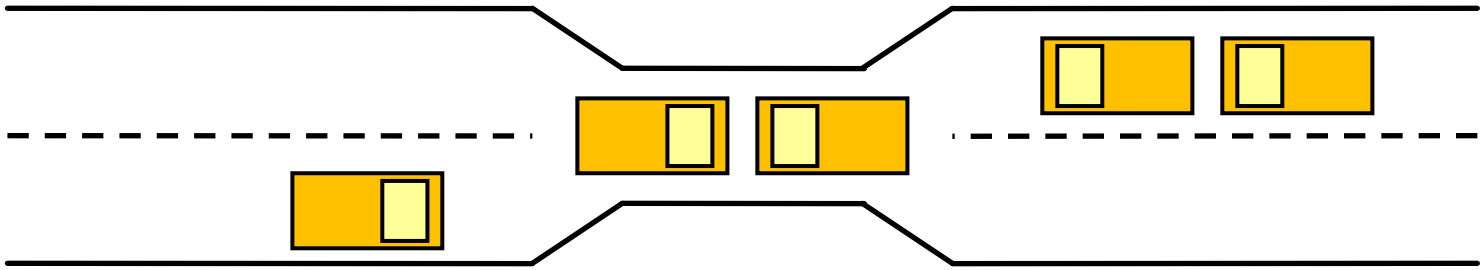
Anomalous behavior due to unexpected critical **dependence on the relative timing** of events.

Source: Free On−Line Dictionary of Computing

Race condition

- Interference caused by
  - Untrusted processes
    - processes running other programs, which “slip in”' other actions between steps of the secure program. These other programs might be invoked by an attacker specifically to cause the problem.
    - **Sequencing problems**
  - Trusted processes
    - conditions caused by processes running the “same” application.
    - **Locking problems (deadlock, livelock, …)**

Deadlocks



- 4 necessary conditions:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

Race condition

- Example
  - Create a file (with default access rights)
  - And then set the access rights of the new file
  - Attacker can get access to the file before the correct permissions are set…

Files – atomic actions…

Sequence of actions is not atomic

- avoid using access(2) to determine if a request should be granted, followed later by open(2)
  - Attacks based on symlinks
- A secure program should instead set its effective id or filesystem id, then make the open call directly

Source: Linux & Unix Secure programming by D. Wheeler

Working with files - examples

- Editor joe
  - the ‘joe’ text editor had a weakness called the ‘DEADJOE’ symlink vulnerability. When joe was exited in a nonstandard way, joe would unconditionally append its open buffers to the file ‘DEADJOE’. This could be exploited by the creation of DEADJOE symlinks in directories where root would normally use joe. In this way, joe could be used to append garbage to potentially sensitive files, resulting in a denial of service and/or unintentional access
  - failure in joe, connection down, …

Example: CVE-2013-4169

Overview

GNOME Display Manager (gdm) before 2.21.1 allows local users to change permissions of arbitrary directories via a symlink attack on /tmp/.X11-unix/.

Description

The GNOME Display Manager (GDM) provides the graphical login screen, shown shortly after boot up, log out, and when user-switching.

A race condition was found in the way GDM handled the X server sockets directory located in the system temporary directory. An unprivileged user could use this flaw to perform a symbolic link attack, giving them write access to any file, allowing them to escalate their privileges to root.

GDM will chown /tmp/.X11-unix to the user and group root, but also changes the permissions to 1777.

Example: CVE-2013-2162

Overview

Race condition in the post-installation script (mysql-server-5.5.postinst) for MySQL Server 5.5 for Debian GNU/Linux and Ubuntu Linux creates a configuration file with world-readable permissions before restricting the permissions, which allows local users to read the file and obtain sensitive information such as credentials.

CVE-2013-5147: iOS Passcode Lock

Description: Passcode Lock in Apple iOS before 7 does not properly manage the lock state, which allows physically proximate attackers to bypass an intended passcode requirement by leveraging a race condition involving phone calls and ejection of a SIM card.

Impact: A person with physical access to the device may be able to bypass the screen lock.

CVE-2013-5035: Race Conditions in HtmlCleaner

Product: Open-Xchange AppSuite / HTMLCleaner

Vulnerability Details:
If multiple requests to save E-Mail as “draft”, or send E-Mail, occur within a very narrow window of time, it is possible that E-Mail content get swapped between requests. The root cause for this is a HTML sanitising library that turned out not to be thread-safe despite it claims to be. Further research showed, that the issue has been introduced with OX 7.2.2 by updating to the latest version of this library (2.2 to 2.5). OX Versions 7.2.1 and earlier are not vulnerable.

CVE-2013-1792: Race Condition in Linux Kernel

There is a race in install_user_keyrings() that can cause a NULL pointer dereference when called concurrently for the same user if the uid and uid-session keyrings are not yet created. It might be possible for an unprivileged user to trigger this by calling keyctl() from userspace in parallel immediately after logging in.

The race window is really small but can be exploited if, for example, thread B is interrupted or preempted after initializing uid_keyring, but before doing setting session_keyring.

This couldn't be reproduced on a stock kernel. However, after placing systemtap probe on 'user->session_keyring = session_keyring;' that introduced some delay, the kernel could be crashed reliably.

Source: https://patchwork.kernel.org/patch/2228431/

CVE-2013-1792

Assume that we have two threads both executing lookup_user_key(), both looking for KEY_SPEC_USER_SESSION_KEYRING.

```
THREAD A                                    THREAD B
==============================              ==============================
                                            ==>call install_user_keyrings();
if (!cred->user->session_keyring)
==>call install_user_keyrings()
                                            ...
                                            user->uid_keyring = uid_keyring;
if (user->uid_keyring)
        return 0;
<==
key = cred->user->session_keyring [== NULL]
                                            user->session_keyring = session_keyring;
atomic_inc(&key->usage); [oops]
```

Source: https://patchwork.kernel.org/patch/2228431/

CVE-2013-0871: Race condition in the ptrace

Race condition in the ptrace functionality in the Linux kernel before 3.7.5 allows local users to gain privileges via a PTRACE_SETREGS ptrace system call in a crafted application.

putreg() assumes that the tracee is not running and pt_regs_access() can safely play with its stack. However a killed tracee can return from ptrace_stop() to the low-level asm code and do RESTORE_REST, this means that debugger can actually read/modify the kernel stack until the tracee does SAVE_REST again.

CVE-2013-2162: Race condition in MySQL

Race condition in the post-installation script (mysql-server-5.5.postinst) for MySQL Server 5.5 for Debian GNU/Linux and Ubuntu Linux creates a configuration file with world-readable permissions before restricting the permissions, which allows local users to read the file and obtain sensitive information such as credentials.

CVE-2012-6095: Race condition in ProFTPD

There is a possible race condition in the handling of the MKD/XMKD FTP commands, when the UserOwner directive is involved, and the attacker is on the same physical machine as a running proftpd:

1. Locally create directory `foo'.
2. In ftp client, send "MKD foo/etc".
3. ProFTPD creates `foo/etc/' directory.
4. Locally move `foo' out of the way.
5. Locally create symlink `foo -> /'.
6. ProFTPD applies UserOwner to `foo/etc', changing ownership of /etc.

The race is the time between when proftpd creates the requested directory (step 3) and when proftpd applies the UserOwner ownership changes (step 6); in that time, a local attack can replace the created directory with a symlink that points to some other directory that the local attacker does not control.

Source: http://bugs.proftpd.org/show_bug.cgi?id=3841

How to do it correctly?

- open file using the modes O_CREAT | O_EXCL and grant only very narrow permissions
  - use umask and/or open's parameters to limit initial access to just the user and user group.
- be prepared that the call to *open* can fail…

time of check–time of use (TOCTOU)

- first open the file and then use the operations on open files
  - i.e. use functions like fchown( ), fstat( ), or fchmod( ) instead of chown(), chgrp(), and chmod() taking filenames.
  - This will prevent the file from being replaced while your program is running…
  - E.g. if you close a file and then use chmod() to change its permissions, an attacker may be able to move or remove the file between those two steps and create a symbolic link to another file (say /etc/passwd). Other interesting files include /dev/zero, which can provide an infinitely-long data stream of input to a program; if an attacker can ``switch'' the file midstream, the results can be dangerous…

Temporary files

- Temporary files are a big issue
  – discusses in a separate lecture

Files: case sensitive?

- Filesystems
  - Case-sensitive (e.g. ext2)
  - Case-insensitive, but case-preserving (e.g. FAT32)
  - Case-insensitive (e.g. FAT16)
- Example
  - Program has a blacklist that prevents users from uploading or downloading the file /etc/ssh_host_key, On a case-insensitive volume, you must also reject someone who makes a request for /etc/SSH_host_key, /ETC/SSH_HOST_KEY, or even /ETC/ssh_host_key.

Scripts & files

- Set the temporary directory ($TMPDIR) environment variable to a safe directory. Even if your script doesn't directly create any temporary files, one or more of the routines you call might create one, which can be a security vulnerability if it's created in an insecure directory.
- Set the umask to restrict access to any files that might be created by routines run by the script.
- Do not redirect output using the operators > or >> to a publicly writable location. These operators do not check to see whether the file already exists, and they follow symbolic links.
- Do not use the test command (or its left bracket ([) equivalent) to check for the existence of a file or other status information for the file before writing to it. Doing so always results in a race condition…

Source: https://developer.apple.com/library/mac/documentation/security/conceptual/SecureCodingGuide/Articles/RaceConditions.html

Critical sections

- Critical section
  - Must be performed exclusively
  - Processes / threads
- Mutual exclusion
  - MutEx (futex, semaphore)

Semaphores in Windows API

- Semaphore-related functions:
  - CreateSemaphore
  - OpenSemaphore
  - ReleaseSemaphore
  - Wait
    - SignalObjectAndWait
    - WaitForSingleObject
    - WaitForSingleObjectEx
    - WaitForMultipleObjects
    - WaitForMultipleObjectsEx
    - MsgWaitForMultipleObjects
    - MsgWaitForMultipleObjectsEx

Semaphore

- YES
  - Wait (x)
  - Signal (x)
- NO
  - Wait (x)
  - Wait (x)
- NO
  - Wait (x)
  - Signal (y)

Secure signal handling

- Signal can come asynchronously any time
- Secure signal processing
  - No system calls
  - Terminate as quickly as possible

> Although there are certain system calls that are safe from within signal handlers, writing a safe signal handler that does so is tricky. The best thing to do is to set a flag that your program checks periodically, and do no other work within the signal handler. This is because the signal handler can be interrupted by a new signal before it finishes processing the first signal, leaving the system in an unpredictable state or, worse, providing a vulnerability for an attacker to exploit.

Source: https://developer.apple.com/library/mac/documentation/security/conceptual/SecureCodingGuide/Articles/RaceConditions.html

Example: CVE-1999-0035

In 1997, a vulnerability was reported in a number of implementations of the FTP protocol in which a user could cause a race condition by closing an FTP connection. Closing the connection resulted in the near-simultaneous transmission of two signals to the FTP server: one to abort the current operation, and one to log out the user. The race condition occurred when the logout signal arrived just before the abort signal.

When a user logged onto an FTP server as an anonymous user, the server would temporarily downgrade its privileges from root to nobody so that the logged-in user had no privileges to write files. When the user logged out, however, the server reassumed root privileges. If the abort signal arrived at just the right time, it would abort the logout procedure after the server had assumed root privileges but before it had logged out the user. The user would then be logged in with root privileges, and could proceed to write files at will. An attacker could exploit this vulnerability with a graphical FTP client simply by repeatedly clicking the “Cancel” button.

Source: https://developer.apple.com/library/mac/documentation/security/conceptual/SecureCodingGuide/Articles/RaceConditions.html

Other file tips

- Before you attempt a file operation, make sure it is safe to perform the operation on that file. For example, before attempting to read a file (but after opening it), you should make sure that it is not a FIFO or a device - special file.
- Just because you can write to a file, that doesn't mean you *should* write to it. For example, the fact that a directory exists doesn't mean you created it, and the fact that you can append to a file doesn't mean you own the file or no one else can write to it.
- Some operations can be done only on certain systems. For example, certain file systems honor setuid files when executed from them and some don't. Be sure you know what file system you're working with and what operations can be carried out on that system.
- Local pathnames can point to remote files. For example, the path /volumes/foo might actually be someone's FTP server rather than a locally-mounted volume. Just because you're accessing something by a pathname, that does not guarantee that it's local or that it should be accessed.
- Remember that users can read the contents of executable binaries just as easily as the contents of ordinary files. For example, the user can run strings(1) to quickly see a list of (ostensibly) human-readable strings in your executable.
- When you fork a new process, the child process inherits all the file descriptors from the parent unless you set the close-on-exec flag. If you fork and execute a child process and drop the child process' privileges so its real and effective IDs are those of some other user (to avoid running that process with elevated privileges), then that user can use a debugger to attach the child process. They can then run arbitrary code from that running process. Because the child process inherited all the file descriptors from the parent, the user now has access to every file opened by the parent process.

Source: https://developer.apple.com/library/mac/documentation/security/conceptual/SecureCodingGuide/Articles/RaceConditions.html

Tools to detect data races

- Static checking
  - LockLint (Sun)
  - Vpara flag of compiler (Sun)
- Runtime checking – simulation based
  - Helgrind (part of Valgrind)
- Runtime checking – execution based
  - Visual Threads (HP)
  - Thread Checker (Intel)
  - Data Race Detection Tool – DRDT (Sun)

Static checking

- Advantages
  - Fast, consumes little memory
  - Analysis does not affect the behaviour of program
  - Can detect potential data races that do not happen in particular run with particular input data set.

Helgrind

- Helgrind = Valgrind tool for detecting synchronisation errors in C/C++ programs that use the POSIX pthreads threading primitives.
- Helgrind can detect three classes of errors:
  1. Misuses of the POSIX pthreads API.
  2. Potential deadlocks arising from lock ordering problems.
  3. Data races - accessing memory without adequate locking or synchronisation

Using helgrind

- C program in sample.c
- Compile the program
  - Use –g to include line numbers
  - gcc -g sample.c -o sample –l pthread
- Run helgrind
  - As a tool of valgrind
  - valgrind --tool=helgrind ./sample
- Analyze the results

Ad 1: Misuses of the POSIX pthreads API.

- unlocking an invalid mutex
- unlocking a not-locked mutex
- unlocking a mutex held by a different thread
- destroying an invalid or a locked mutex
- recursively locking a non-recursive mutex
- deallocation of memory that contains a locked mutex
- passing mutex arguments to functions expecting reader-writer lock arguments, and vice versa
- when a POSIX pthread function fails with an error code that must be handled
- when a thread exits whilst still holding locked locks
- calling pthread_cond_wait with a not-locked mutex, an invalid mutex, or one locked by a different thread
- inconsistent bindings between condition variables and their associated mutexes
- invalid or duplicate initialisation of a pthread barrier
- initialisation of a pthread barrier on which threads are still waiting
- destruction of a pthread barrier object which was never initialised, or on which threads are still waiting
- waiting on an uninitialised pthread barrier

Example: missing unlock

```
#include <pthread.h>

int main ( void )
{
   pthread_mutex_t mx1;
   pthread_mutex_init( &mx1, NULL );
   pthread_mutex_lock( &mx1 );

   return 0;
}
```

Example: missing unlock

```
==7131== Helgrind, a thread error detector
==7131== Copyright (C) 2007-2012, and GNU GPL'd, by OpenWorks LLP et al.
==7131== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==7131== Command: ./a.out
==7131==
==7131== ---Thread-Announcement------------------------------------------
==7131==
==7131== Thread #1 is the program's root thread
==7131==
==7131== ----------------------------------------------------------------
==7131==
==7131== Thread #1: Exiting thread still holds 1 lock
==7131==    at 0x414C52C0: _Exit (in /usr/lib/libc-2.16.so)
==7131==
==7131==
==7131== For counts of detected and suppressed errors, rerun with: -v
==7131== Use --history-level=approx or =none to gain increased speed, at
==7131== the cost of reduced accuracy of conflicting-access information
==7131== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Ad 2: Potential deadlocks arising from lock ordering problems.

- Helgrind builds a directed graph indicating the order in which locks have been acquired in the past. When a thread acquires a new lock, the graph is updated, and then checked to see if it now contains a cycle. The presence of a cycle indicates a potential deadlock involving the locks in the cycle.
- Helgrind will choose two locks involved in the cycle and show you how their acquisition ordering has become inconsistent. It does this by showing the program points that first defined the ordering, and the program points which later violated it.

Example: deadlock

```
#include <pthread.h>

static pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

static void *t1(void *v)
{
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2);
        pthread_mutex_unlock(&m1);
        pthread_mutex_unlock(&m2);

        return 0;
}

static void *t2(void *v)
{
        pthread_mutex_lock(&m2);
        pthread_mutex_lock(&m1);
        pthread_mutex_unlock(&m1);
        pthread_mutex_unlock(&m2);

        return 0;
}
```

```
int main()
{
        pthread_t a, b;

        pthread_mutex_lock(&m1);
        pthread_mutex_unlock(&m1);

        pthread_create(&a, NULL, t1, NULL);
        pthread_create(&b, NULL, t2, NULL);

        pthread_join(a, NULL);
        pthread_join(b, NULL);

        return 0;
}
```

Example: deadlock

```
==7487== Thread #3: lock order "0x804A02C before 0x804A044" violated
==7487==
==7487== Observed (incorrect) order is: acquisition of lock at 0x804A044
==7487==    at 0x400B0BF: pthread_mutex_lock (hg_intercepts.c:495)
==7487==    by 0x80485FE: t2 (hg02.c:19)
==7487==    by 0x400AD68: mythread_wrapper (hg_intercepts.c:219)
==7487==    by 0x41614AFE: start_thread (in /usr/lib/libpthread-2.16.so)
==7487==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
==7487==
==7487==  followed by a later acquisition of lock at 0x804A02C
==7487==    at 0x400B0BF: pthread_mutex_lock (hg_intercepts.c:495)
==7487==    by 0x804860A: t2 (hg02.c:20)
==7487==    by 0x400AD68: mythread_wrapper (hg_intercepts.c:219)
==7487==    by 0x41614AFE: start_thread (in /usr/lib/libpthread-2.16.so)
==7487==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
==7487==
==7487== Required order was established by acquisition of lock at 0x804A02C
==7487==    at 0x400B0BF: pthread_mutex_lock (hg_intercepts.c:495)
==7487==    by 0x80485C1: t1 (hg02.c:9)
==7487==    by 0x400AD68: mythread_wrapper (hg_intercepts.c:219)
==7487==    by 0x41614AFE: start_thread (in /usr/lib/libpthread-2.16.so)
==7487==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
==7487==
==7487==  followed by a later acquisition of lock at 0x804A044
==7487==    at 0x400B0BF: pthread_mutex_lock (hg_intercepts.c:495)
==7487==    by 0x80485CD: t1 (hg02.c:10)
==7487==    by 0x400AD68: mythread_wrapper (hg_intercepts.c:219)
==7487==    by 0x41614AFE: start_thread (in /usr/lib/libpthread-2.16.so)
==7487==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
```

Ad 3: Data races

- A data race happens, or could happen, when two threads access a shared memory location without using suitable locks or other synchronisation to ensure single-threaded access. Such missing locking can cause obscure timing dependent bugs.
- Ensuring programs are race-free is one of the central difficulties of threaded programming…

Example: data race

```
#include <pthread.h>

static int shared;

static void *th(void *v)
{
        shared++;
        return 0;
}

int main()
{
        pthread_t a, b;

        pthread_create(&a, NULL, th, NULL);
        pthread_create(&b, NULL, th, NULL);

        pthread_join(a, NULL);
        pthread_join(b, NULL);

        return 0;
}
```

Example: data race

```
==2740== ------------------------------------------------------------
==2740==
==2740== Possible data race during read of size 4 at 0x6009E0 by thread #3
==2740== Locks held: none
==2740==    at 0x40057C: th (hg04.c:7)
==2740==    by 0x4C2D0D4: mythread_wrapper (hg_intercepts.c:219)
==2740==    by 0x4E37850: start_thread (in /lib64/libpthread-2.12.so)
==2740==    by 0x6BE16FF: ???
==2740==
==2740== This conflicts with a previous write of size 4 by thread #2
==2740== Locks held: none
==2740==    at 0x400585: th (hg04.c:7)
==2740==    by 0x4C2D0D4: mythread_wrapper (hg_intercepts.c:219)
==2740==    by 0x4E37850: start_thread (in /lib64/libpthread-2.12.so)
==2740==    by 0x5DE06FF: ???
==2740==
==2740== ------------------------------------------------------------
```

Valgrind: DRD tool

- drd, a thread error detector

```
==7691== Thread 3:
==7691== Conflicting load by thread 3 at 0x0804a024 size 4
==7691==    at 0x8048533: th (in /home/zriha/h/a.out)
==7691==    by 0x400B17C: vgDrd_thread_wrapper (drd_pthread_intercepts.c:355)
==7691==    by 0x41614AFE: start_thread (in /usr/lib/libpthread-2.16.so)
==7691==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
==7691== Allocation context: BSS section of /home/zriha/h/a.out
==7691== Other segment start (thread 2)
==7691==    at 0x400D5F0: pthread_mutex_unlock (drd_pthread_intercepts.c:703)
==7691==    by 0x400B171: vgDrd_thread_wrapper (drd_pthread_intercepts.c:236)
==7691==    by 0x41614AFE: start_thread (in /usr/lib/libpthread-2.16.so)
==7691==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
==7691== Other segment end (thread 2)
==7691==    at 0x414FE659: madvise (in /usr/lib/libc-2.16.so)
==7691==    by 0x41614C30: start_thread (in /usr/lib/libpthread-2.16.so)
==7691==    by 0x415020ED: clone (in /usr/lib/libc-2.16.so)
==7691==
...
```

Excercise

1. Play with Helgrind

Homework

- Helgrind
  - Create a program in C/C++ that contains 5 type of errors that are reported by Helgrind. Then correct the errors.
    - Submit 2 source codes and 2 Helgrind reports