

1. The problem caused by bugs such as memory leak and concurrency errors.

Memory Leak: A memory leak usually occurs when a computer program do not release the memory which is not needed anymore. The most directed problem caused by memory leak is the reduced performance. Because of the reducing available memory, a program will run much slower than it should be. In the worst case, the program may fails, or the system slows down vastly.

Dangling pointer: A dangling pointer is a pointer which does not point to a valid object of the appropriate type. A dangling pointer usually occurs when the reference the pointer points to is deleted. For example, if a dangling pointer is used for writing things to memory, some other data structure may be corrupted.

Access to unauthorized memory: Access to unauthorized memory means a pointer try to access the memory which is not authorized by the program. In some situation, a bad access will make the program fails. In the other situation, a bad access will lead to some unpredictable outcomes.

Concurrency errors: A concurrency error occurs due to the incorrect synchronization safeguards in multi-threaded program. A concurrency error may cause data corruption, security vulnerability and incorrect behavior. Besides, a concurrency error is hard to find, which needs a lot of time, and it is also difficult to test.

Deadlock: Deadlock is one kind of concurrency error. It occurs when two or more processes are never able to process because each is waiting for the others to do something. A deadlock may cause the program stop processing.

Race condition: A race condition is another kind of concurrency error. It occurs when concurrent accesses to a shared variable and at least one of the access is a write. Race condition will lead to some unpredictable problems and reduce the reliability of the process.

2. Comparison between garbage collection approach and a C/C++ style memory management approach.

Garbage Collection: Garbage collector is an automatic memory management. It attempts to reclaim memory occupied by the objects which are no longer needed. Compared with C/C++ style memory management approach, garbage collector do not need programmer to manage memory leaks manually. However, as tradeoff for convenience, garbage collector does have some cons. First, it cannot handle circular references. Second, it has significant memory and time overhead for memory management.

C/C++ style memory management: Unlike languages such as JAVA, C and C++ do not have garbage collector. For C and C++, programmer needs to check memory management manually. Manual memory management has some pros. First, it does not have overhead since the program does not need extra memory and time to check memory leak. Second, it can process all kinds of memory leak. However, manual memory management requires a lot of time for programmer to solve all the memory problems. Therefore it is quite inconvenient.

3. Describe the different concurrency errors with examples. That is, provide a code sample which when executed will result in the error.

Deadlock:

As mentioned before, deadlock occurs when two or more threads are waiting for each other therefore none of them is able to continue processing. In the following code, **threadA** locked m1 and is waiting for m2. At the same time **threadB** locked m2 and is waiting for m1. Therefore none of can continuously process and a deadlock is formed.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t m1, m2;

void *threadA(void *d)
{
    printf("threadA entered");
    pthread_mutex_lock(&m1);        //m1 is locked
    pthread_mutex_lock(&m2);        //m2 is locked and wait for m2 to be unlocked
    pthread_mutex_unlock(&m2);      //unlock m2
    pthread_mutex_unlock(&m1);      //unlock m1
}

void *threadB(void *d)
{
    printf("threadB entered");
    pthread_mutex_lock(&m2);        //m2 is locked
    pthread_mutex_lock(&m1);        //m1 is locked and wait for
    pthread_mutex_unlock(&m1);      //unlock m1
    pthread_mutex_unlock(&m2);      //unlock m2
}

int main(int argc, char* argv[])
{
    pthread_t t1,t2;
    pthread_mutex_init(&m1,0);      //initialize m1
    pthread_mutex_init(&m2,0);      //initialize m2
    pthread_create(&t1,NULL,threadA,0); //create threadA
    pthread_create(&t2,NULL,threadB,0); //create threadB
    pthread_join(t1,(void**)0);
    pthread_join(t2,(void**)0);
    pthread_mutex_destroy(&m1);
    pthread_mutex_destroy(&m2);
    return 0;
}
```

Race condition: as mentioned above, a race condition occurs concurrent threads try to access one variable/area in the same time and at least one of them is a write. The code below [1] is an example of race condition. In the code below, we do not know when the variable **var** adds in the **child_fn** thread and when **var** is adds in the **parent** thread, therefore a concurrency error occurs.

```
#include <pthread.h>
int var = 0;
void* child_fn ( void* arg ) {
    var++;
    return NULL;
}
int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++;
    pthread_join(child, NULL);
    return 0;
}
```

4. Description of the different tools and how they find the bugs described earlier.

Electric Fence:

Electric fence is a memory error detector, which mainly helps to detect two common bugs:

1. access to unauthorized memory
2. dangling pointer (both are described above)

Unlike other malloc() debuggers, Electric fence can detect both read and write, and it can find the exact instructions that cause the problems.

How Electric Fence finds the bugs: Electric fence finds the bugs by replacing normal **malloc()** function with a version which allocates the requested memory and (usually) allocates a section of memory immediately after this, which the process is not allowed to access. Because of the extra memory, the kernel can halt the process immediately with a segmentation fault if the program tries to access the memory.

Memcheck from Valgrind:

Memcheck is another memory debugger, it can detect the following problems that are common in the C and C++ program:

1. Access memory you shouldn't
2. Use undefined values
3. Incorrect free of heap memory
4. Overlap **src** and **dst** pointers in **memcpy** and related functions.
5. Pass a fishy (presumably negative) value to the **size** parameter of a memory allocation function
6. Memory leaks

How Memcheck finds the bugs: Memcheck has two public tables: Valid-Address Map and Valid-Value Map.

Valid-Value Map: For every byte in the address space of the process, there are 8 corresponding bits in the valid-value map. Besides, for every register in CPU, there is also a corresponding bit in the valid-value map. These bits can record whether the byte and register is valid and the value of them.

Valid-Address Map: For every byte in the address space of the process, there is a corresponding bit. These bits record whether the address can be accessed (read/written).

When the process wants to read/write some bytes in the address space, we need to check the bit in Valid-Address Map which corresponds to it. If the A-Bit shows it is invalid, memcheck will report that a read/write error occurs.

Memcheck's core is a virtual CPU environment. When some bytes in the memory are loaded into real CPU, the bytes stored in Valid-Value Map which correspond to it will be loaded into the virtual CPU. When the value in register is used to calculate the address, memcheck will check its V-Bit. If its V-Bit has not been initialized yet, then memcheck will issue an error.

Helgrind for checking synchronization errors.

Helgrind is also a Valgrind tool which is for detecting synchronization errors in C, C++ and Fortran programs that use POSIX pthreads threading primitives. Helgrind can detect three kinds of errors.

1. Misuses of the POSIX pthread API
2. Potential deadlocks arising from ordering problem
3. Data race conditions

How Helgrind finds the bugs:

For misuses of the POSIX pthread API: Helgrind intercepts calls to many POSIX pthreads functions, and is therefore able to report on various common problems.

For potential deadlocks: Helgrind monitors the order in which thread acquires locks, which allows to detect potential deadlocks which could arise from the formation of cycles of locks.

For data race conditions: Helgrind monitors all accesses to memory locations and check to see if two accesses ordered by happens-before relation. If two accesses doesn't have a happens-before relationship, a race occurs.

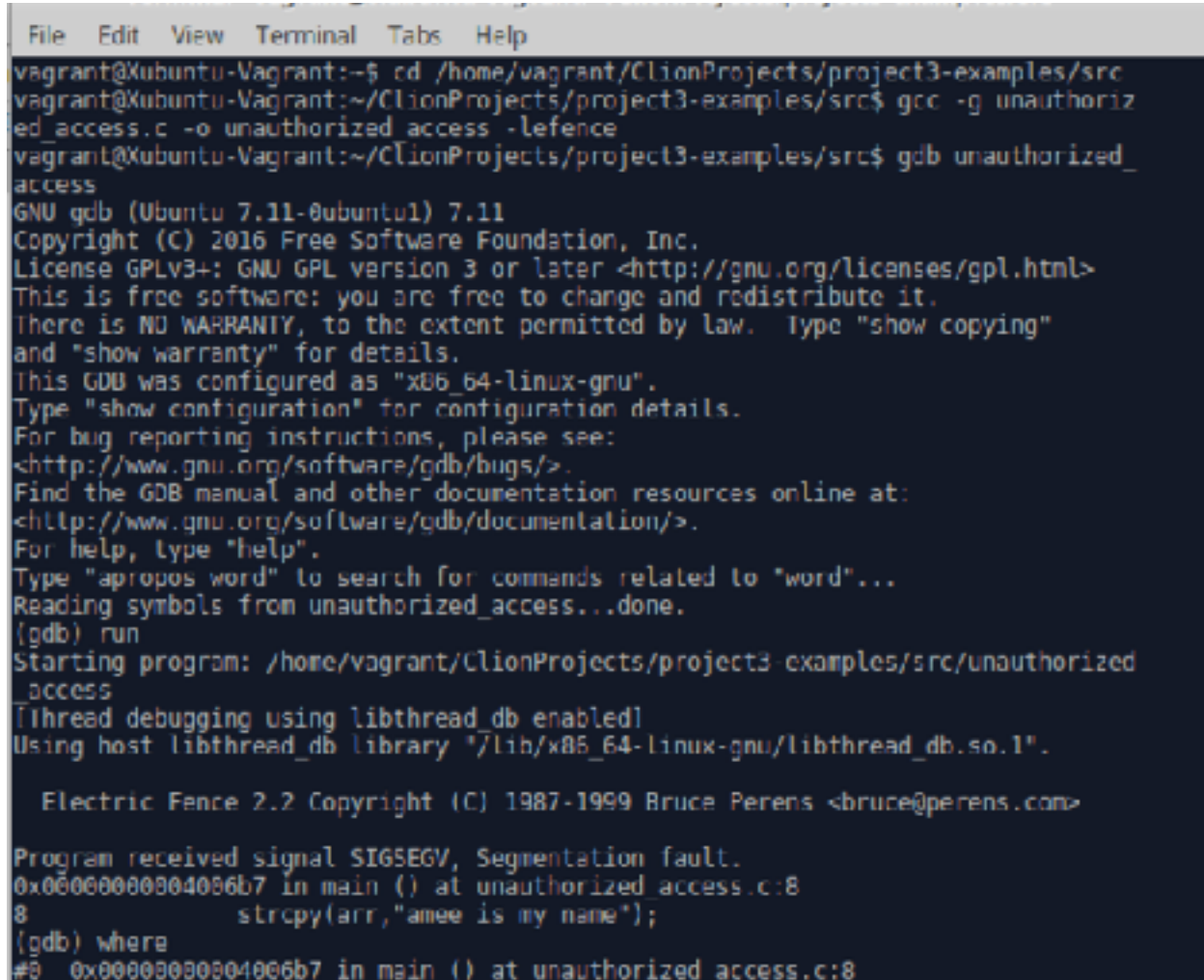
5. Describe the usage of these tools using example.

Electric Fence: Use -lefence argument to the linker or put the path-name for libefence.a in the linker's command line. After the code file is compiled. Use debugger such as GDB to locate the erroneous statement.

1. **access to unauthorized memory:** the code [2] below is clearly an access to unauthorized memory bug In the following code, variable **arr** only have memory of five characters but the program try to give it a string which is longer than 5 characters. Therefore it is an access to unauthorized memory.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char *arr;    int i;
    arr = (char *)malloc(sizeof(char)*5);
    strcpy(arr,"amee is my name");
    return 0;
}
```

In order to use Electric fence, I use the command “**gcc -g unauthorized_access.c -o unauthorized_access -lefence**” and then I use GDB to locate the error. From the result, we can see that there is a segmentation fault (which is an access to unauthorized memory). It is occurred in main() at unauthorized_memory.c “strcpy(arr, “amee is my name”);”



```
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~$ cd /home/vagrant/ClionProjects/project3-examples/src
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gcc -g unauthorized_access.c -o unauthorized_access -lefence
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gdb unauthorized_access
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from unauthorized_access...done.
(gdb) run
Starting program: /home/vagrant/ClionProjects/project3-examples/src/unauthorized_access
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

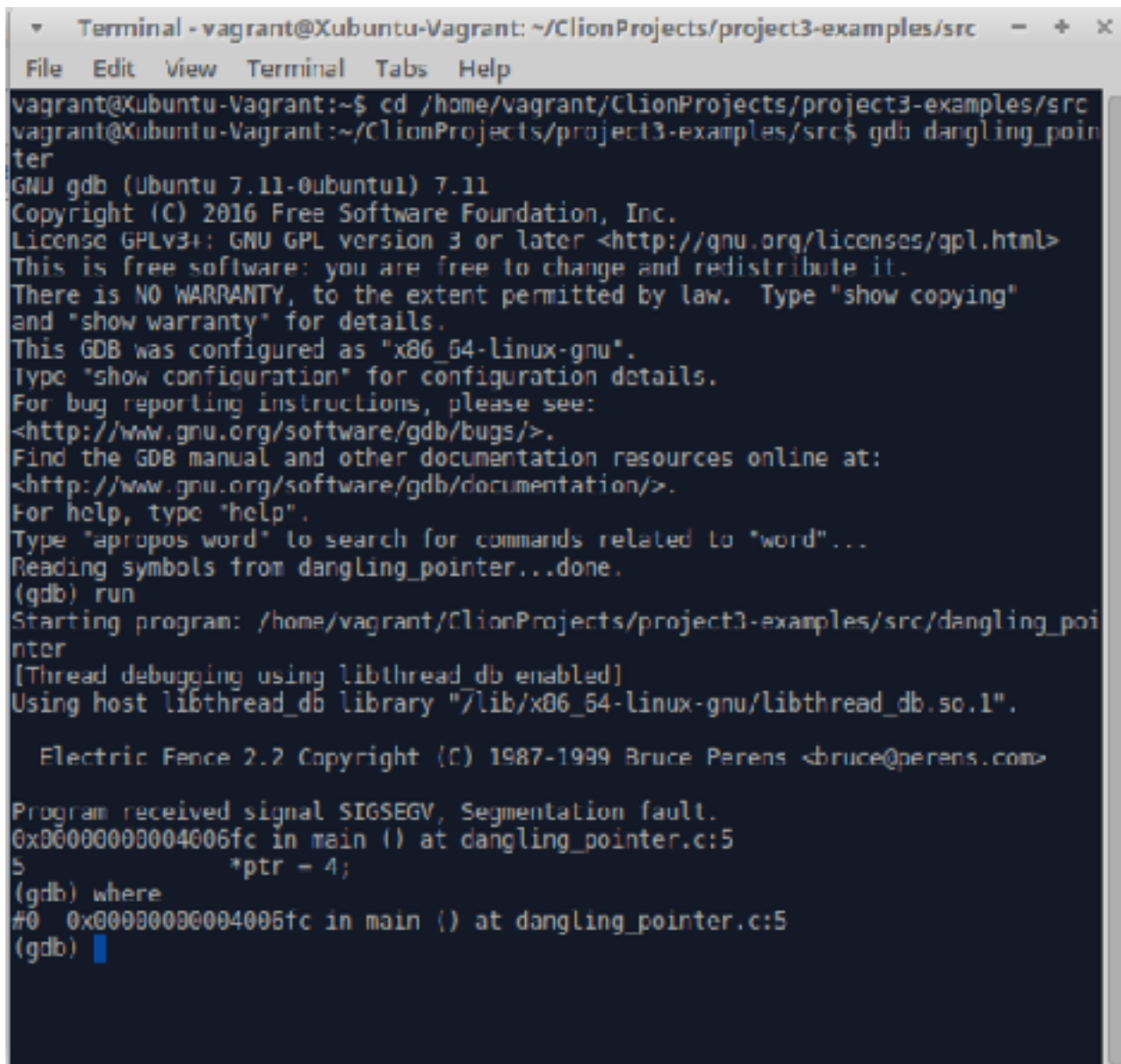
Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
0x000000004006b7 in main () at unauthorized_access.c:8
8      strcpy(arr,"amee is my name");
(gdb) where
#0  0x000000004006b7 in main () at unauthorized_access.c:8
```


2. dangling pointer: the code below is clearly a dangling pointer since the pointer **ptr** is released ***ptr** is invoked.

```
#include <stdlib.h>
int main( void ) {
    int *ptr = malloc(sizeof(int));
    free(ptr);
    *ptr = 4;
}
```

By using Electric Fence, we can see the result below. It is clear that there is a segmentation fault (which is a dangling pointer). It is occurred in main at dangling_pointer.c “*ptr = 4;”



```
Terminal - vagrant@Xubuntu-Vagrant: ~/ClionProjects/project3-examples/src - + x
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~$ cd /home/vagrant/ClionProjects/project3-examples/src
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gdb dangling_pointer
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from dangling_pointer...done.
(gdb) run
Starting program: /home/vagrant/ClionProjects/project3-examples/src/dangling_pointer
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

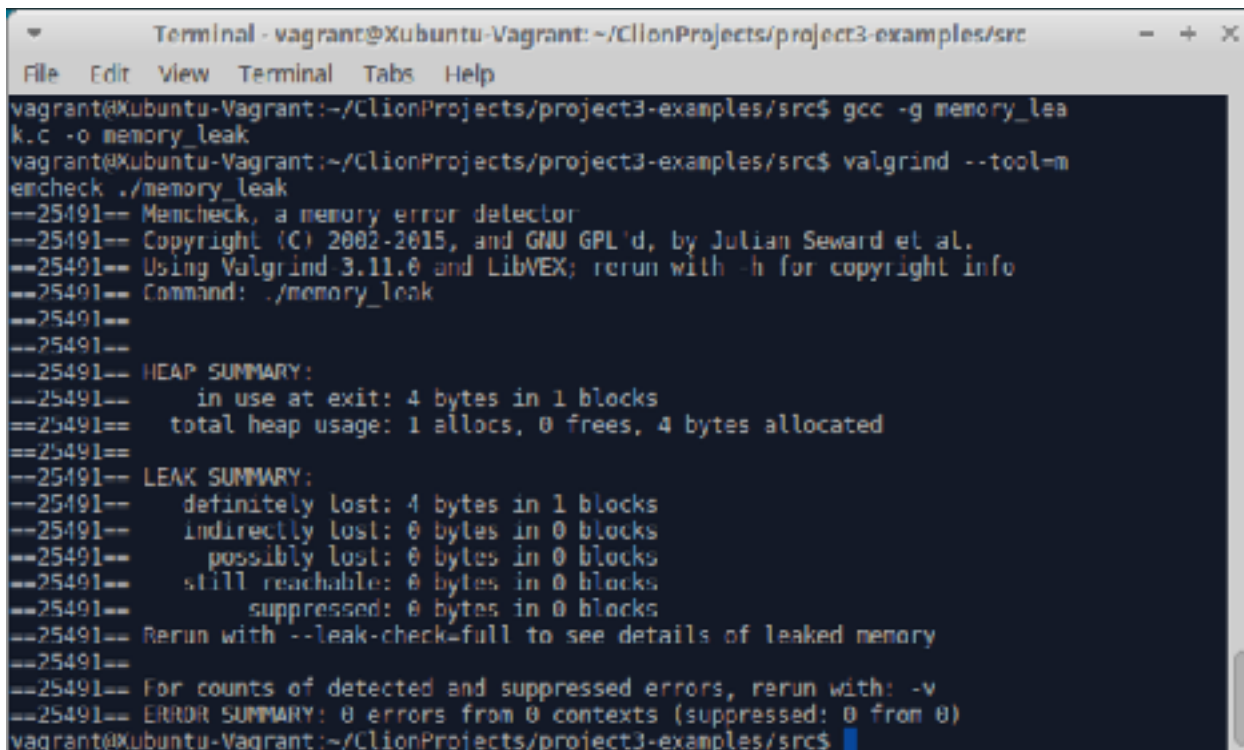
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005fc in main () at dangling_pointer.c:5
5      *ptr = 4;
(gdb) where
#0  0x00000000004005fc in main () at dangling_pointer.c:5
(gdb) 
```

Memcheck: After the code is compiled. Use “**Valgrind --tool==memcheck**” + the executable to call the memcheck to check the code.

1. Memory Leak: the code below clearly has a memory leak since the pointer **ptr** is not needed and is not freed.

```
#include <stdlib.h>
int main ( void ) {
    int *ptr = malloc(sizeof(int));
}
```

By using Memcheck, we can see the result below. From heap summary, there's 1 allocs and 0 frees. From leak summary, we can see there's a definitely lost: 4 bytes in 1 blocks, which means there is a memory leak.



```
Terminal - vagrant@Xubuntu-Vagrant: ~/ClionProjects/project3-examples/src
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gcc -g memory_leak.c -o memory_leak
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ valgrind --tool=memcheck ./memory_leak
==25491== Memcheck, a memory error detector
==25491== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==25491== Using Valgrind 3.11.0 and LibVEX; rerun with -h for copyright info
==25491== Command: ./memory_leak
==25491==
==25491==
==25491== HEAP SUMMARY:
==25491==    in use at exit: 4 bytes in 1 blocks
==25491==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==25491==
==25491== LEAK SUMMARY:
==25491==    definitely lost: 4 bytes in 1 blocks
==25491==    indirectly lost: 0 bytes in 0 blocks
==25491==    possibly lost: 0 bytes in 0 blocks
==25491==    still reachable: 0 bytes in 0 blocks
==25491==    suppressed: 0 bytes in 0 blocks
==25491== Rerun with --leak-check=full to see details of leaked memory
==25491==
==25491== For counts of detected and suppressed errors, rerun with: -v
==25491== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$
```

2. Uninitialized Value: the code below has an uninitialized value. Since the pointer **p** doesn't point to any char but the it is assigned to the variable **c**.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *p;
    char c = *p;
    printf("c is %c\n",c);
    return 0;
}
```

By using Memcheck, we can see the result below. Memcheck states that there is a use of uninitialized value of size 8, which was created by a stack allocation. Besides, Memcheck also mentioned which line in the code occurs the uninitialized value.

```
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gcc -g uninstalled_value.c -o
uninstalled_value
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ valgrind --track-origins=yes
--tool=memcheck ./uninstalled_value
==25597== Memcheck, a memory error detector
==25597== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==25597== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==25597== Command: ./uninstalled_value
==25597==
==25597== Use of uninitialised value of size 8
==25597==    at 0x400532: main (uninstalled_value.c:7)
==25597==    Uninitialised value was created by a stack allocation
==25597==    at 0x400526: main (uninstalled_value.c:5)
==25597==
==25597== Invalid read of size 1
==25597==    at 0x400532: main (uninstalled_value.c:7)
==25597== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==25597==
==25597==
==25597== Process terminating with default action of signal 11 (SIGSEGV)
==25597== Access not within mapped region at address 0x0
==25597==    at 0x400532: main (uninstalled_value.c:7)
==25597== If you believe this happened as a result of a stack
==25597== overflow in your program's main thread (unlikely but
==25597== possible), you can try to increase the size of the
==25597== main thread stack using the --main-stacksize= flag.
==25597== The main thread stack size used in this run was 8388608.
==25597==
==25597== HEAP SUMMARY:
==25597==    in use at exit: 0 bytes in 0 blocks
==25597== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==25597==
==25597== All heap blocks were freed -- no leaks are possible
==25597==
==25597== For counts of detected and suppressed errors, rerun with: -v
==25597== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$
```

3. Unauthorized Access: the code below has an unauthorized access, the pointer **p** is only allocated with 1 byte but **c** is tried to access the memory which is not authorized to be accessed.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *p = malloc(1);
    *p = 'a';
    char c = *(p+1);
    free(p);
    return 0;
}
```

By using Memcheck, we can see the result below. Memcheck tells us that there is an invalid read of size 1 and at the same time tells where the error occurs.

```
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gcc -g unauthorized_access2.c
-o unauthorized_access2
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ valgrind --tool=memcheck ./unauthorized_access2
==25688== Memcheck, a memory error detector
==25688== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==25688== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==25688== Command: ./unauthorized_access2
==25688==
==25688== Invalid read of size 1
==25688==    at 0x400507: main (unauthorized_access2.c:8)
==25688==    Address 0x5203041 is 0 bytes after a block of size 1 allocated
==25688==    at 0x4C2008F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25688==    by 0x400577: main (unauthorized_access2.c:6)
==25688==
==25688== HEAP SUMMARY:
==25688==    in use at exit: 0 bytes in 0 blocks
==25688==    total heap usage: 1 allocs, 1 frees, 1 bytes allocated
==25688==
==25688== All heap blocks were freed -- no leaks are possible
==25688==
==25688== For counts of detected and suppressed errors, rerun with: -v
==25688== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$
```

Helgrind: Like Memcheck, Helgrind is also a part of Valgrind. Therefore the way to use Helgrind is also similar to that of Memcheck. After the code is compiled, we use “**Valgrind — tool=Helgrind**” + the executable.

Deadlock: the code which contains a deadlock is presented before. And below is the result of using Helgrind to check the deadlock. Helgrind tells us that Thread #3: lock order violated, which is a deadlock. In addition, Helgrind also shows us which instructions occurs a deadlock and at which place the lock is first observed.

```
Terminal: vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ valgrind --tool=helgrind ./deadlock
==25812== Helgrind, a thread error detector
==25812== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==25812== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==25812== Command: ./deadlock
==25812==
==25812== ---Thread Announcement-----
==25812==
==25812== Thread #3 was created
==25812==    at 0x56381E: clone (clone.S:74)
==25812==    by 0x40109: create_thread (createthread.c:101)
==25812==    by 0x4171C3: pthread_create@@GLIBC_2.2.5 (pthread_create.c:670)
==25812==    by 0x434887: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x40095D: main (in /home/vagrant/ClionProjects/project3-examples/src/deadlock)
==25812==
==25812== Thread #3: lock order "0x6010A0 before 0x6010A0" violated
==25812==
==25812== Observed (incorrect) order is: acquisition of lock at 0x6010A0
==25812==    at 0x4321BC: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x400911: threadB (in /home/vagrant/ClionProjects/project3-examples/src/deadlock)
==25812==    by 0x434086: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x4171F9: start_thread (pthread_create.c:333)
==25812==
==25812== Followed by a later acquisition of lock at 0x6010A0
==25812==    at 0x4171BC: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x40088B: threadB (in /home/vagrant/ClionProjects/project3-examples/src/deadlock)
==25812==    by 0x4171F9: start_thread (pthread_create.c:333)
==25812==
==25812== Required order was established by acquisition of lock at 0x6010E0
==25812==    at 0x4321BC: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x40090B: threadA (in /home/vagrant/ClionProjects/project3-examples/src/deadlock)
==25812==
==25812== by 0x434086: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812== by 0x4171F9: start_thread (pthread_create.c:333)
==25812==
==25812== Lock at 0x6010E0 was first observed
==25812==    at 0x43608a: pthread_mutex_lock (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x40090E: main (in /home/vagrant/ClionProjects/project3-examples/src/deadlock)
==25812== Address 0x6010E0 is 0 bytes inside data symbol "n1"
==25812==
==25812== Lock at 0x6010A0 was first observed
==25812==    at 0x43608a: pthread_mutex_lock (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25812==    by 0x40091D: main (in /home/vagrant/ClionProjects/project3-examples/src/deadlock)
==25812== Address 0x6010A0 is 0 bytes inside data symbol "n2"
==25812==
==25812== threadA enteredthreadB entered==25812==
==25812== For counts of detected and suppressed errors, rerun with: -v
==25812== Use --history-level=approx or --none to gain increased speed, at
==25812== the cost of reduced accuracy of conflicting-access information
==25812== DRD006 SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$
```

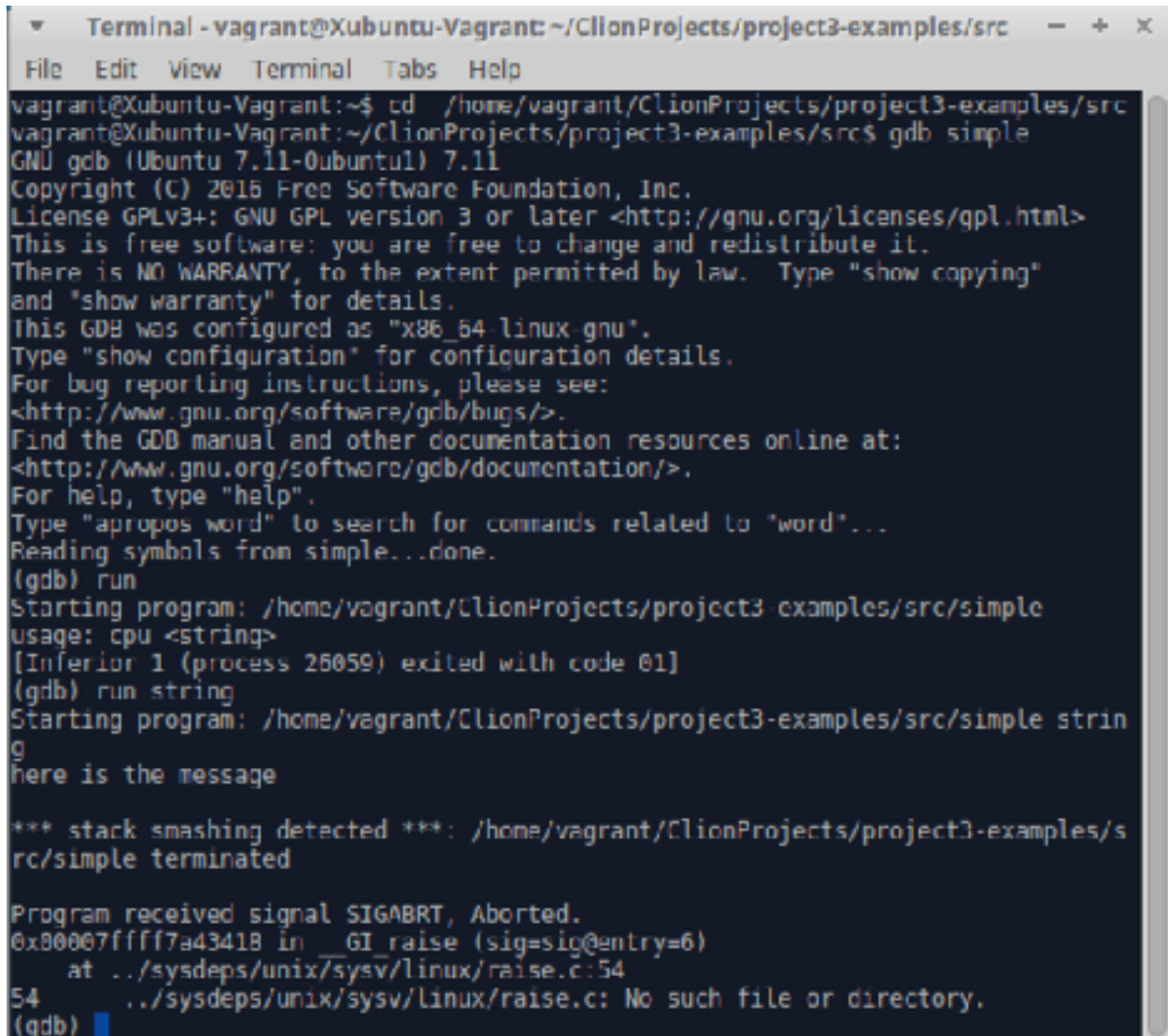

Race Condition: Same as deadlock, the code of race condition is also presented above. Below is the result of using Helgrind to detect race condition error. From the result, we can see that Helgrind tell us there is a possible data race during write of size 4 by thread #1. It also show us the address of the data race, which is 0x060104C. Besides, Helgrind also tells us that there is no lock held so there is no deadlock

```
Terminal - vagrant@Xubuntu-Vagrant: ~/ClionProjects/project3-examples/src
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ valgrind --tool=helgrind ./race_condition
==25838== Helgrind, a thread error detector
==25838== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==25838== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==25838== Command: ./race_condition
==25838==
==25838== ---Thread-Announcement-----
==25838==
==25838== Thread #1 is the program's root thread
==25838==
==25838== ---Thread-Announcement-----
==25838==
==25838== Thread #2 was created
==25838==   at 0x516381E: clone (clone.S:74)
==25838==   by 0x4E46109: create_thread (createthread.c:102)
==25838==   by 0x4E47EC3: pthread_create@GLIBC_2.2.5 (pthread_create.c:679)
==25838==   by 0x4C34887: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25838==   by 0x400795: main (in /home/vagrant/ClionProjects/project3-examples/src/race_condition)
==25838==
==25838==
==25838== Possible data race during read of size 4 at 0x60104C by thread #1
==25838== Locks held: none
==25838==   at 0x400795: main (in /home/vagrant/ClionProjects/project3-examples/src/race_condition)
==25838==
==25838== This conflicts with a previous write of size 4 by thread #2
==25838== Locks held: none
==25838==   at 0x4006C7: child_fn (in /home/vagrant/ClionProjects/project3-examples/src/race_condition)
==25838==   by 0x4C340B6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25838==   by 0x4E476F9: start_thread (pthread_create.c:333)
==25838== Address 0x60104c is 0 bytes inside data symbol "var"
==25838==
==25838==
==25838== Possible data race during write of size 4 at 0x60104C by thread #1
==25838== Locks held: none
==25838==   at 0x40079F: main (in /home/vagrant/ClionProjects/project3-examples/src/race_condition)
==25838==
==25838== This conflicts with a previous write of size 4 by thread #2
==25838== Locks held: none
==25838==   at 0x4006C7: child_fn (in /home/vagrant/ClionProjects/project3-examples/src/race_condition)
==25838==   by 0x4C340B6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==25838==   by 0x4E476F9: start_thread (pthread_create.c:333)
==25838== Address 0x60104c is 0 bytes inside data symbol "var"
==25838==
==25838==
==25838== For counts of detected and suppressed errors, rerun with: -v
==25838== Use --history-level=approx or =none to gain increased speed, at
==25838== the cost of reduced accuracy of conflicting-access information
==25838== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$
```

6. Analysis of the code in project 3 example

simple.c:

There's no problem detected by Helgrind and Memcheck. However, when using Electric Fence, we can see that there is a buffer overflow problem in the **simple.c**.



```
Terminal - vagrant@Xubuntu-Vagrant: ~/ClionProjects/project3-examples/src
File Edit View Terminal Tabs Help
vagrant@Xubuntu-Vagrant:~$ cd /home/vagrant/ClionProjects/project3-examples/src
vagrant@Xubuntu-Vagrant:~/ClionProjects/project3-examples/src$ gdb simple
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from simple...done.
(gdb) run
Starting program: /home/vagrant/ClionProjects/project3-examples/src/simple
usage: cpu <string>
[Inferior 1 (process 25059) exited with code 01]
(gdb) run string
Starting program: /home/vagrant/ClionProjects/project3-examples/src/simple string
here is the message

*** stack smashing detected ***: /home/vagrant/ClionProjects/project3-examples/s
rc/simple terminated

Program received signal SIGABRT, Aborted.
0x00007ffff7a43418 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) █
```

From the result above, we can see that after running the executable, the GDB debugger asks us to run `<string>`, and after we ran `<string>`, a stack smashing detected, which implied that the code has a buffer overflow.

malloc.c:

There's no error detected by Helgrind. However, by using Memcheck and Electric Fence, we can see that there is an error.

Electric Fence:

```
Terminal - vagrant@Xubuntu-Vagrant: ~/ClionProjects/project3-examples/src
File Edit View Terminal Tabs Help
(gdb) run
Starting program: /home/vagrant/ClionProjects/project3-examples/src/malloc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
the current location of brk is 0x602000
value of variable b is 0
the location of brk after malloc is 0x602000
real allocated bytes = 0. We asked for 40
address of i[0]:0x7ffff7ee4fec: value stored is 1
address of i[1]:0x7ffff7ee4ff0: value stored is 1
address of i[2]:0x7ffff7ee4ff4: value stored is 1
address of i[3]:0x7ffff7ee4ff8: value stored is 1
address of i[4]:0x7ffff7ee4ffc: value stored is 1
there is no guarantee that j will begin immediately after i.
address of j[0]:0x7ffff7ee2fec: value stored is 2
address of j[1]:0x7ffff7ee2ff0: value stored is 2
address of j[2]:0x7ffff7ee2ff4: value stored is 2
address of j[3]:0x7ffff7ee2ff8: value stored is 2
address of j[4]:0x7ffff7ee2ffc: value stored is 2
Now we check the value of i.
address of i[0]:0x7ffff7ee4fec: value stored is 1
address of i[1]:0x7ffff7ee4ff0: value stored is 1
address of i[2]:0x7ffff7ee4ff4: value stored is 1
address of i[3]:0x7ffff7ee4ff8: value stored is 1
address of i[4]:0x7ffff7ee4ffc: value stored is 1

Program received signal SIGSEGV, Segmentation fault.
0x000000000040093b in main () at malloc.c:58
58      printf("address of i[%d]:%p: value stored is %d\n", counter, i +
    counter, i[counter]);
(gdb) where
#0  0x000000000040093b in main () at malloc.c:58
(gdb)
```

Electric fence shows that there is a segmentation fault, which mean there must be an error about memory management. Also, electric fence shows us where the error is **malloc.c:58**.

Memcheck:

Furthermore, Memcheck tells us that there is a memory leak and an uninitialized value in the code

```
==26087== HEAP SUMMARY:
==26087==    in use at exit: 60 bytes in 3 blocks
==26087==    total heap usage: 4 allocs, 1 frees, 1,084 bytes allocated
==26087==
==26087== LEAK SUMMARY:
==26087==    definitely lost: 60 bytes in 3 blocks
==26087==    indirectly lost: 0 bytes in 0 blocks
==26087==    possibly lost: 0 bytes in 0 blocks
==26087==    still reachable: 0 bytes in 0 blocks
==26087==    suppressed: 0 bytes in 0 blocks
==26087== Rerun with --leak-check=full to see details of leaked memory
==26087==
==26087== For counts of detected and suppressed errors, rerun with: -v
==26087== Use --track-origins=yes to see where uninitialised values come from
==26087== ERROR SUMMARY: 13 errors from 9 contexts (suppressed: 0 from 0)
vagrant@Xubuntu-Vagrant: ~/ClionProjects/project3-examples/bin$
```

```
==26087== Use of uninitialised value of size 8
==26087==    at 0x4E8472B: _itoa_word (_itoa.c:179)
==26087==    by 0x4E880EC: vfprintf (vfprintf.c:1631)
==26087==    by 0x4E8F848: printf (printf.c:33)
==26087==    by 0x4008B4: main (in /home/vagrant/ClionProjects/project3-examples
/bin/malloc_ex)
==26087==
```

7. Present a summary of lessons learned.

First, I learnt different types of bugs and how they are occurred:

- Memory Leak
- Access to unauthorized memory
- Dangling Pointers
- Deadlock
- Data race condition

Second, I learnt how to use different tools to find these bugs:

- Memcheck: used to find bugs about memory management
- Electric Fence: used to find bugs about access to unauthorized memory and dangling pointer
- Helgrind: used to find bugs about concurrency error

Reference:

[1]: code source: <http://mcs.une.edu.au/doc/valgrind/html/hg-manual.html>

[2]: code source: http://elinux.org/Electric_Fence