Memory Leak Detection

Under the guidance of: Dr. Yann-Hang Lee
Team members: Tarun Vyas, Rahul Thakkar
What is a Memory leak ???

• A **memory leak** is a particular kind of unintentional memory consumption by a computer program where the program fails to release memory when no longer needed.

• A memory leak may happen if an object is stored in the memory and cannot be accessed by the running code in any condition.

• Memory leaks are the contributing factors to Software Aging which implies progressive performance degradation or sudden crash of software system due to exhaustion of OS resources.

• Memory leaks are common errors in programming languages like C/C++ which do not provide automatic garbage collection techniques.

• Memory leak reports are usually submitted as unit test results!
Consequences of Memory leaks

• In user level process running in modern OS, the memory consumed by the application will be released when the application terminates; which may not be serious in systems with large memories.

• But Memory leaks are much more serious, for example in cases:
  o The program is supposed to run for long times. Eg: Back-end Servers.
  o Embedded devices which may be left running for years such as: Network switches and hubs.
  o Programs where new memory is allocated for continuing tasks such as : rendering the frames on the screen in an video device.

• Prevalence of these memory leak bugs has led the development of various debugging tools. Good examples of such tools are:
  1. Valgrind
  2. IBM Rational Purify
  3. Memwatch
  4. Insure++
Electric Fence

• Electric fence (Efence) is a memory (malloc) debugger which helps to detect two common programming bugs:
  1. Accessing the memory region beyond the boundary.
  2. Touching the memory region which has already been released.

• Electric fence uses the virtual memory hardware to place an inaccessible memory page before and after the memory allocated.

• Thus, when the code line will try to access the memory bound, then hardware will issue a segmentation fault SIGSEGV signal and will terminate the process at the offending instruction, giving the exact location of error.

• To use Efence, the application must be linked with libefence.a library. If Electric fence is installed on the machine, then -lefence argument can be supplied to the linker.
• An example to use electric fence is shown below:

```c
extern int EF_ALIGNMENT;
int main()
{
    char *m;
    int i = 0;
    EF_ALIGNMENT = 0; // Set this variable to detect over run with byte precision w/o compiler byte padding
    m = (char *)malloc(sizeof(char)*10);
    if (NULL == m)
    {
        printf("No memory\n");
        exit(-1);
    }
    for (i=0; i<= 10; i++)
    {
        m[i] = 'a';  //over-run for 11th byte – efence will issue segmentation fault
    }
    printf("Done with error\n");
    return 0;
}
```

• gcc -o output test.c -lefence
Limitations of Electric fence

• It acts as only malloc debugger which means that it doesn't debug dynamically allocated memory for its uninitialized use.

• Efence for a given platform is not a generic tool. User must specify through the global flags like EF_ALIGNMENT, EF_PROTECT_BELOW etc. as to what kind of errors Efence would detect.

• Efence can't work when the malloc enhancement libraries are used.

• The source code of Efence is not thread safe.

• Efence uses at least two virtual pages per allocation, thus the memory overhead is very high. 'Purify does a much more thorough job than Electric Fence, and does not have the huge memory overhead' –cited in the man page of Efence.
Memcheck from Valgrind

• Valgrind originally was a tool for memory debugging, memory leaks & program profiling. It has evolved as a generic framework for dynamic binary instrumentation.

• Valgrind in essence is a process virtual machine i.e it runs as a normal application inside the host OS and supports a single process.

• Valgrind works on the executable binary and dynamically recompiles it to translate to a simpler form called as Intermediate Representation (IR). This part is done by the valgrind core.

• Memcheck – most popular tool in the Valgrind suite works on this IR and then adds instrumentation code around the instructions and tracks the validity and addressability of the memory blocks
Meta-data

• **A (Addressability) bits:**
  o Every memory byte is shadowed with a single 'A' bit. It checks if the client program legitimately accesses the memory byte.
  
  o 0 represents an un-addressable byte and 1 represents addressable byte. These values will get updated as the memory block gets allocated and freed, and checked every time that block is accessed.

• **V (Validity) bits:**
  o Every register and memory byte is shadowed with 8 V bits. It indicates whether the value of bits is defined (initialized or derived from other defined values).
  
  o 0 represents a defined bit and 1 represents undefined bit. Every value writing operation in the shadowed with other operation that updates these V bits.
Implementation of shadow memory (for 32 bit architecture)

Entries PM[1] and PM[2] which have been written point to corresponding SM whereas remaining entries point to NO-ACCESS DSM
Shadow Memory data structures

- The Memcheck's main shadow memory data structure is primarily a two-level table, designed for 32 bit address space (4GB).

- The implementation for 64 bit address space is different. The address space is divided into 64K chunks each of 64KB.

- The primary map (PM) is an array of 64K entries and each entry is a pointer pointing to Secondary map structure (SM) which contains the A bits and V bits.

- There is also distinguished secondary map (DSM) which is marked entirely unaddressable. All the PM entries initially point to it.

```c
typedef struct {
    U8 abits8[8192]; // 8K A Bytes/64K A bits
    U8 vbits8[65536]; // 64K V bytes
} SM;

SM * PM[65536]; // Primary map //covering 4GB
```
Single byte load and store

- The Memcheck tool is implemented on top of fundamental functions to load and store the individual shadow memory bytes.
- These functions are not the exact implementation in release versions of valgrind which have optimizations added but when tool is put in debugging mode, it falls back to these.

- Let's see the functions required for loading shadow bytes:
  1. Memcheck uses the high 16 bits of the address to find the relevant SM within the PM.
     ```c
     SM* get_SM_for_reading(Addr a) {
         Return PM[a >> 16]; // [31-16] bits of a to locate SM
     }
     ```
  2. Memcheck uses the lower 16 bits of the address to find the A bits and V bits within the SM. Loads are done with the following function:
     ```c
     get_abit_and_vbits8. Extracting the V bits is straightforward but extracting A bits require some shifting and masking.
     ```
• void get_abit_and_vbits8(Addr a, /*Out */ Uw * abit /*Out */ Uw * vbits8 )
  {
    SM * sm = get_SM_for_reading(a);
    // 13 bits of address used to determine 1 byte of 8 A bits
    U8 abits8 = sm->abits8[(a & 0xffff) >> 3];
    // last 3 bits used to determine the A bit in the byte
    *abit = 0x01 & (abits8 >> (a & 0x7));
    *vbits8 = sm->vbits8[a & 0xffff];
  }

• On every access to any address the load functions will be called. Similarly, store functions will change the A bit and the V bits.

• Also, it can be inferred that every shadow memory byte has 257 meaningful states. The V bits will be considered for validity only when the A bit says that the memory byte (V bits) is addressable.

• This gives Memcheck, the power to be precise on the bit level and thus the false positive and false negative rate is significantly low.
• Memcheck also records the location of every heap block with trace information in a hash table. In the end, the table is scanned for all the blocks to find any pointers references along with the shadow bit details, to detect any memory leaks and bad frees.
Limitations of Valgrind - Memcheck

- Memcheck is a dynamic analysis tool. The code will be checked for only the parts which it falls through. The rest of the code is not checked.
- Memcheck has a big memory overhead and hence the code runs much slower. The mean slowdown factor using various benchmarks in SPEC2006 benchmark suite is between 22 to 23.5 as per the analysis done in Ref[2].
- Memcheck catches the errors only below the stack.
- It is incapable of detecting the static array bound errors.

```c
stack_test()
{
    // This test provides an example of reading/writing inappropriate
    // areas on the stack. Note that valgrind only catches errors below
    // the stack (so in this example, valgrind won't catch the problem)
    int i;
    int * ptr = &i;
    ptr[8] = 7;  // Error, writing to a bad location on stack --not caught
    i = ptr[15];  // Error, reading from a bad stack location -- not caught
    i = ptr[-10];  // Error will be caught by Valgrind
}
```
IBM Rational Purify

- Is a run-time error detection tool
- It parses the object code and inserts its own verification code.
- It maintains a table where each memory byte is tracked by having two additional bits for each byte.
- Thus, it can define four states or regions for each memory byte which are:

Initially the heap and stack are marked red.

Purify also inserts guard zones before and after each allocated block and statically allocated data item to prevent illegal read or write.
Purify classifies memory errors into four types:

- Using un-initialized (by our program) memory: It can be a read or a memory transfer. These two are termed **UMR and UMC** purify.

- Illegal access to memory: Illegal access to memory not owned by a program is pretty easy when pointer arithmetic is employed. Reading such memory may lead to segmentation fault or core dump.

  ```c
  void InvalidPointerRead() {
    int *ipr = (int *) malloc(4 * sizeof(int));
    int i, j;
    i = *(ipr - 1000); j = *(ipr + 1000);
    free(ipr);
  }
  ```

- Read/write beyond the stack: When a called function returns a pointer and the callee tries to de-reference that pointer, it is actually trying to read from beyond the stack boundaries since the stack shrinks after function return.
Buffer overruns: is caused when a program writes more data than it is allowed to on a buffer allocated on the stack. It is a well known security issue.

```c
void genABRandABW() {
    const char *name = "This is a buffer overrun";
    char *str = (char*) malloc(10);
    strncpy(str, name, 10);
    str[11] = '\0';
    printf("%s\n", str);
}
```

Finally we have MLK (memory leak errors), FIH (freeing invalid heap) errors that can be detected by purify.

```c
int *pi;
void foo() {
    pi = (int*) malloc(8*sizeof(int));
    free(pi);
}
void main() {
    pi = (int*) malloc(4*sizeof(int));
    foo();
    pi[0] = 10;
}
```
• Using memory area after releasing:

```c
int i,*p,*temp;
p = malloc(10 * sizeof(int));
temp = p;
for (i=0; i< 5; i++)
{
    temp = temp + i;  // temp points to intermediate location
}
free(p);
/* temp points in between the memory area which is already freed 
and hence any further access to temp may result in disastrous results */
printf("%d\n", temp[i]);
```

• int main()
{
    char *p;
    int i;
    for (i = 0 ;i < 5; i++)
    {
        p = test(i+1);
        printf("Returned string is %s\n", p);
    }
    free(p); // will free only the last allocated value
    printf("Done\n");
    return 0;
}

char *test(int j)
{
    int i;
    char * temp;
    temp = malloc(sizeof(char)*j);
    for (i=0; i< j; i++)
        temp[i] = 'a';
    return temp;
}
• The above short examples in isolation are pretty simple to detect, but in complex codes these errors may become very difficult to track. Observe the below sample code:

```c
int main (int argc, char *argv[]) {
    Char *lower;
    Lower = to_lower (argv[1]);
    while (*lower)
        putchar (*{lower++});
    printf("\n\n");
    Return 0;
}
```

```c
char *to_lower (const char *str)
{
    char *l = strdup (str);
    char *c;
    for (c = l; *c; c++)
    {
        if (isupper(*c))
            *c = tolower(*c);
    }
    return l;
}
```

• So, on one glance we don't find any memory leak associated with this code. When we run tools such as Valgrind or Purify, the memory leaks are visible.
The Valgrind report for the above piece of code when input is "NAME1":

HEAP SUMMARY:
==3623==  in use at exit: 6 bytes in 1 blocks
==3623==  total heap usage: 1 allocs, 0 frees, 6 bytes allocated

==3623==  6 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3623==  at 0x4024F20: malloc (vg_replace_malloc.c:236)
==3623==  by 0x40AC07F: strdup (strdup.c:43)

==3623==  by 0x80484E4: to_lower (tough_example.c:12)
==3623==  by 0x8048557: main (tough_example.c:27)

The definition of strdup function:

```c
char *strdup (const char *s) {
    char *d = malloc (strlen (s) + 1); // Space for length plus nul
    if (d == NULL) return NULL; // No memory
    strcpy (d, s); // Copy the characters
    return d; // Return the new string
}
```
Limitations of Purify

- The following piece of code will report false-positive for Purify but works well on Valgrind (since Valgrind works on bit-precision level and Purify on byte):

```c
void set_bit(int* arr, int n){
    arr[n / 32] |= (1 << (n % 32));
}

int get_bit(int* arr, int n){
    return 1 & (arr[n/32] >> (n % 32));
}

void main(){
    int *arr = malloc(10*sizeof(int));
    set_bit(arr,10);
    printf("%d\n",get_bit(arr,10));
}
```
Kernel Memory Leaks

• As stated many a times, kernel is a huge data structure and since, the memory used here is freed only at re-boot, things can go very bad over a period of time.

• The tools that we discussed in the presentation, report only those errors caused by a user-space program.

• We have a patch by Catalin Marinas, kmemleak which was introduced in the kernel 2.3 onwards.

• For kmemleak, the CONFIG_DEBUG_KMEMLEAK option in the /usr/src/linux-headers-<kernel_version>/.config file has to be turned on to enable leak debugging.

• Rebuild the kernel

• A kernel threads does a scan every 10 mins and the leaks can be viewed in /sys/kernel/debug/kmemleak
• All the memory allocations are stored in a priority-search tree along with other details such as size and stack trace.

• Initially all the memory objects are marked white or orphans (which have no pointer to them so that they can be freed)

• Upon a scan all the objects are looked up for a valid pointer in the tree, and if one is found, they are marked as gray.

• The remaining white objects if any are reported in the `kmemleak` file.

• Due to temporary pointer storage in CPU registers and stack on SMPs, some of the leaks are only transient ones. (To avoid this they have set a minimum age limit for an object to be reported as a memory leak)
Overhead in Valgrind

- Since Valgrind employs Dynamic Binary Instrumentation, overhead is inevitable.
- We have used Mozilla Firefox binary (well known for memory leaks) to evaluate the overhead incurred by Valgrind.
- We ran the binary first with memcheck tool and measured the time taken by Valgrind to report.

```
valgrind --tool=memcheck --leak-check=yes --time-stamp=yes ./firefox-bin

==00:00:00:06.817 4113== LEAK SUMMARY:
 ==00:00:00:06.817 4113== definitely lost: 0 bytes in 0 blocks
 ==00:00:00:06.817 4113== indirectly lost: 0 bytes in 0 blocks
 ==00:00:00:06.817 4113== possibly lost: 261 bytes in 6 blocks
 ==00:00:00:06.817 4113== still reachable: 0 bytes in 0 blocks
 ==00:00:00:06.817 4113== suppressed: 0 bytes in 0 blocks
==00:00:00:06.817 4113==
==00:00:00:06.817 4113== For counts of detected and suppressed errors, rerun with: -v
==00:00:00:06.817 4113== ERROR SUMMARY: 6 errors from 6 contexts
(suppressed: 164 from 9)
```
• Next we ran the binary on Nullgrind (which just invokes Valgrind but doesn’t do any instrumentation) and again measured the time taken.

• The instrumentation overhead we observed was: 5.017x slowdown by memcheck.

• The above test was performed on Linux 2.6.32 – 31 – generic on Intel(R) Core(TM) 2 Duo CPU T7250@2GHZ and 2GB memory.
Project work

- In our project work we will run 4-5 benchmarks from the SPEC 2006 benchmark suite on Valgrind and Purify to calculate the overhead incurred by these two tools.
- We will also run an older version of Mozilla Firefox and analyze the report generated by memcheck.
References

1. ‘Using Valgrind to detect undefined value errors with bit precision’. Nicholas Nethercote, Julian Seward.
2. ‘How to Shadow every byte of memory used by a program’. Nicholas Nethercote, Julian Seward.
7. http://lwn.net/Articles/187193/
8. Precise detection of memory leaks' by Jonas Maebe Michiel Ronsse Koen De Bosschere, Ghent University, ELIS Department.
Thank you !!!