Using the Debugger

EECS 678 Staff

Debugger

- ## What is a debugger?
  - A powerful tool that supports examination of a program at runtime

- ## How it works?
  - Creates additional symbol tables that enables tracking program behavior and relating it back to the source files

- ## Examples
  - gdb, sdb, dbx etc.

GDB

- gdb is a debugger for C & C++ code
- Capabilities
  - Execute a program
  - Stop the execution of a program using breakpoints:
    - At any line unconditionally in any source file
    - At the beginning of any function
    - At any point in the program when a certain condition is met
    - …
  - Examine various types of information regarding the runtime execution environment
    - Value of any variable which is in the current scope
    - Condition of the runtime stack (call graph, stack frame etc.)
    - Condition of the hardware registers (stack pointer, frame pointer, program counter etc.)

GDB

- Capabilities (Cont.)
  - Stop execution when a variable changes (watch points)
  - Call any function (which is callable in the current scope) at any point during execution
  - Examine the content of memory cells

Quiz

- Let's say your program consists of three source files:
  - main.c
  - functions.c
  - functions.h

While debugging 'main.c', is it possible to call a function which is declared and defined in 'functions.c'? Why or why not?

Compilation for Debugging

- Code must be compiled with the **–g** option

  ```
  >> gcc -g -o main.c functions.c my_program.o
  ```

- Debugging is always performed on source files

- Optimizations must be disabled if the program is being compiled for debugging
  - Optimizations can remove variables, modify code organization and cause jumps in program execution while debugging

Valgrind

- A heavyweight tool for dynamically catching hard-to-detect errors in programs
  - Only checks code that it runs
- Valgrind is a virtual machine and quite a bit slower than normally executing program in native environment
  - Translates binaries into an intermediate representation, performs some modifications and recompiles the transformations at runtime as needed
  - Allows it to instrument code with safe guards and profiling tools
- Valgrind's main tool is a memory checker that detects memory errors in programs
- Also has a few other useful tools such as cache and branch miss-prediction profilers, a heap profiler, a multi-thread data race detector, and more

Valgrind

- Memory checker (memcheck) helps find:
  - Memory leaks
  - Usage of uninitialized variables
    - Often the cause of non-deterministic behavior in single threaded software
- Bad frees of heap blocks
  - Double free
  - Freeing unallocated memory
- Accesses to unallocated memory
- Accesses to invalid stack and heap addresses
  - Freed heap addresses
  - Out of bounds heap addresses
  - Out of bounds stack addresses

Lab - 01

- Your task is to fix the problems present in the sample program (see the tar file provided with this lab)

- Currently, the program generates a segmentation fault

- Debugging the program requires the usage of both gdb and valgrind

Memcheck

- [jrobinson@localhost Development]$ **valgrind ./valgrind_test**
- ==2606== Memcheck, a memory error detector
- ==2606== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
- ==2606== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
- ==2606== Command: ./test
- ==2606==
- ==2606== **Conditional jump or move depends on uninitialised value(s)**
- ==2606==   **at 0x40056A: main (test.c:9)**
- ==2606==
- ==2606==
- ==2606== HEAP SUMMARY:
- ==2606==   in use at exit: 1,552 bytes in 202 blocks
- ==2606==   total heap usage: 202 allocs, 0 frees, 1,552 bytes allocated
- ==2606==
- ==2606== LEAK SUMMARY:
- ==2606==   **definitely lost: 800 bytes in 100 blocks**
- ==2606==   **indirectly lost: 700 bytes in 100 blocks**
- ==2606==   **possibly lost: 10 bytes in 1 blocks**
- ==2606==   **still reachable: 42 bytes in 1 blocks**
- ==2606==   suppressed: 0 bytes in 0 blocks
- ==2606== **Rerun with --leak-check=full to see details of leaked memory**
- ==2606==
- ==2606== For counts of detected and suppressed errors, rerun with: -v
- ==2606== **Use --track-origins=yes to see where uninitialised values come from**
- ==2606== ERROR SUMMARY: 101 errors from 1 contexts (suppressed: 0 from 0)

```c
12 #include <stdlib.h>
13
14 void* still_reachable;
15 void* possibly_lost;
16
17 int main() {
18   int uninitialized_variable; // This variable is never given a value.
19
20   for (; uninitialized_variable < 100; uninitialized_variable++) {
21     void** definitely_lost = (void**) malloc(sizeof(void*)); // allocate a
22                                                              // pointer on the
23                                                              // heap.
24
25     *definitely_lost = (void*) malloc(7); // Give the pointer something else to
26                                           // point to on the heap. This will be
27                                           // indirectly lost.
28   }
29
30   // At this point, definitely_lost is out of scope and we can no longer free
31   // it. The pointer pointed to by definitely_lost is indirectly lost since we
32   // were only able to reach the pointer through definitely_lost.
33
34   still_reachable = malloc(42); // This value is never freed but is pointed to
35                                 // in the global scope at program completion.
36
37   possibly_lost = malloc(10);
38   possibly_lost += 4; // This is similar to still reachable except there is a
39                       // pointer pointing to the middle of the allocated block
40                       // but nothing points to the front of the block. This is
41                       // very odd behavior and usually is a memory leak (but not
42                       // always).
43   return 0;
44 }
```

valgrind_test.c – a bad program that compiles without warnings (gcc -Wall -g valgrind_test.c)

Memcheck

- Rerun with "--leak-check=full" and "--track-origins=yes" as suggested by memcheck.

```
[jrobinson@localhost Development]$ valgrind --leak-check=full --track-origins=yes ./test
==16850== Memcheck, a memory error detector
==16850== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==16850== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==16850== Command: ./test
==16850==
==16850== Conditional jump or move depends on uninitialised value(s)
==16850==    at 0x40056A: main (test.c:9)
==16850==  Uninitialised value was created by a stack allocation
==16850==    at 0x400536: main (test.c:6)
==16850==
==16850==
==16850== HEAP SUMMARY:
==16850==     in use at exit: 1,552 bytes in 202 blocks
==16850==   total heap usage: 202 allocs, 0 frees, 1,552 bytes allocated
==16850==
==16850== 10 bytes in 1 blocks are possibly lost in loss record 1 of 4
==16850==    at 0x4C28C50: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==16850==    by 0x400586: main (test.c:26)
==16850==
==16850== 1,500 (800 direct, 700 indirect) bytes in 100 blocks are definitely lost in loss record 4 of 4
==16850==    at 0x4C28C50: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==16850==    by 0x400549: main (test.c:10)
==16850==
==16850== LEAK SUMMARY:
==16850==    definitely lost: 800 bytes in 100 blocks
==16850==    indirectly lost: 700 bytes in 100 blocks
==16850==      possibly lost: 10 bytes in 1 blocks
==16850==    still reachable: 42 bytes in 1 blocks
==16850==         suppressed: 0 bytes in 0 blocks
==16850== Reachable blocks (those to which a pointer was found) are not shown.
==16850== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==16850==
==16850== For counts of detected and suppressed errors, rerun with: -v
==16850== ERROR SUMMARY: 103 errors from 3 contexts (suppressed: 0 from 0)
```

Memcheck

- Rerun with "--show-leak-kinds=all" appended to the options to see the still reachable blocks

- Valgrind is not perfect
  - It will only catch errors that only occur during a program run
  - If an error inducing code is not reached or specific input conditions that trigger an error are not met, then Valgrind will not detect it in that run
  - Importance of thorough unit tests

- Custom memory allocators can confuse Memcheck

Valgrind Example

- When we run ./bugs you may notice that Mars has a non-zero amount of bugs on it. That isn't correct
    - The current bug population of Earth is about: 1480000000000000000
    - The current bug population of Mars is about: 140728482299848
    - The current bug population of Venus is about: 0

- Lets try to throw Valgrind at the problem to see if something weird is happening with memory

- Run Valgrind on the bugs executable:
```
>> valgrind ./bugs
```

Valgrind Example - Uninitialized Variables

- We see a lot of output but at the top we see a few statements about uninitialized variables

```
==25224== Use of uninitialised value of size 8
==25224==    at 0x4E7E0BB: _itoa_word (in /usr/lib64/libc-2.22.so)
==25224==    by 0x4E8262F: vfprintf (in /usr/lib64/libc-2.22.so)
==25224==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==25224==    by 0x40089A: main (bugs.c:108)
==25224==
==25224== Conditional jump or move depends on uninitialised value(s)
==25224==    at 0x4E7E0C5: _itoa_word (in /usr/lib64/libc-2.22.so)
==25224==    by 0x4E8262F: vfprintf (in /usr/lib64/libc-2.22.so)
==25224==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==25224==    by 0x40089A: main (bugs.c:108)
==25224==
==25224== Conditional jump or move depends on uninitialised value(s)
==25224==    at 0x4E826AD: vfprintf (in /usr/lib64/libc-2.22.so)
==25224==    by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==25224==    by 0x40089A: main (bugs.c:108)
```

- It looks like an uninitialized variable is being passed around the printf() call graph quite a bit

Valgrind Example - Uninitialized Variables

- On one of the lines near the bottom notice that Valgrind suggests to use the option --track-origin=yes

- Lets run valgrind again with that and see what we get

$ valgrind --track-origins=yes ./bugs

==26830== Memcheck, a memory error detector
==26830== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==26830== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==26830== Command: ./bugs
==26830==
The current bug population of Earth is about: 1480000000000000000
==26830== Use of uninitialised value of size 8
==26830==   at 0x4E7E0BB: _itoa_word (in /usr/lib64/libc-2.22.so)
==26830==   by 0x4E8262F: vfprintf (in /usr/lib64/libc-2.22.so)
==26830==   by 0x4E88D48: printf (in /usr/lib64/libc-2.22.so)
==26830==   by 0x40089A: main (bugs.c:108)
==26830== Uninitialised value was created by a stack allocation
==26830==   at 0x4007DE: main (bugs.c:85)

- The uninitialized variable is an allocation by the stack is a function call. The function is "main" in this example

Valgrind Example - Uninitialized Variables

- This is a bit challenging since there were several variables in main that are allocated at the same time in the `bug_info struct`

- For this you have to look at the source code to see what is uninitialized

- A way to fix this without question is to initialize each variable in the function right after it was declared

- We were having trouble with the Mars print out so lets look at that first

  ```
  printf("The current bug population of Mars is about: %zu\n",
         bug_info.num_bugs_on_mars);  // 0
  ```

- Check to see if there are any initializations done on this variable above this line. If not add "bug_info.num_bugs_on_mars = 0;" underneath the Storage struct declaration of "bug_info"

- Valgrind should now no longer report the uninitialized variables

GDB Example

- To debug the bugs executable with GDB run GDB from the command line

  `>> gdb ./bugs`

- Once in GDB type "r" (run) and hit enter

- This will run your executable until the segmentation fault and then pause the program

- These examples assume you are only using GDB and have never seen or studied the source code before

GDB Example – Backtrace

- Type "bt" (backtrace) and press enter to see a stack trace of the program
- The output should look something like this:

```
#0  0x00007ffff7a6737c in vfprintf () from /lib64/libc.so.6
#1  0x00007ffff7a6ed49 in printf () from /lib64/libc.so.6
#2  0x0000000000400717 in echo (strs=0x7fffffffdd08) at bugs.c:53
#3  0x0000000000400736 in echo (strs=0x7fffffffdd00) at bugs.c:56
#4  0x0000000000400736 in echo (strs=0x7fffffffdcf8) at bugs.c:56
#5  0x0000000000400736 in echo (strs=0x7fffffffdcf0) at bugs.c:56
#6  0x0000000000400736 in echo (strs=0x7fffffffdce8) at bugs.c:56
#7  0x0000000000400736 in echo (strs=0x7fffffffdce0) at bugs.c:56
#8  0x0000000000400736 in echo (strs=0x7fffffffdcd8) at bugs.c:56
#9  0x0000000000400736 in echo (strs=0x7fffffffdcd0) at bugs.c:56
#10 0x00000000004008ed in main (argc=1, argv=0x7fffffffde08) at bugs.c:124
```

- The top two functions are heavily used library functions printf() and vprintf(), so we can assume the segmentation fault was caused by the author of bugs.c misusing them rather than something inside them
- Also we can clearly see that echo is a recursive function

GDB Example – Stack Navigation

- The #2 stack frame above appears to be the last one of the function **'echo'** which was invoked from **'bugs.c'** before the segmentation fault
- Moreover it looks like it didn't reach as far (line 53) as the rest of the echo functions (line 56)
- To view this stack frame we need to navigate to it. That can be accomplished with "f 2" (frame 2). Another way would be to use the "up" command twice
- You should see the following statements on your terminal after performing the navigation

  (gdb) f 2
  #2 0x0000000000400717 in echo (strs=0x7fffffffdd08) at bugs.c:53
  53   printf("%s ", *strs);

- There is a problem in this printf() statement. Since we ruled out that printf() itself was the problem, lets check the arguments to printf()

GDB Example – Printing Types

- printf() expects a format string as it's first argument of type `char*`, we can see that the first argument is "`%s`"

- **To figure out what the type of a variable is we can use the "ptype"** command like so:
  ```
  (gdb) ptype "%s "
  type = char [4]
  ```

- `char[4]` is just a fancy way for GDB to say that the size of the array containing the string at compile time. This is equivalent to a `char*`. Which is what we wanted

GDB Example – Printing Types

- The second argument to `printf()` is determined by what is in the format string. Since `%s` is the first format specifier then the second argument to `printf()` should be of type `char*`

    (gdb) ptype strs
    type = char **
    (gdb) ptype *strs
    type = char *

- bugs.c is passing `*strs` which is of type `char*` which is what `printf()` expects

- We can also print the type signatures of functions defined in our program:

    (gdb) ptype echo
    type = void (char **)

- The types passed to `printf()` as arguments are correct so lets try to print the value of strs

GDB Example – Printing Values

- To print the value of strs, we can use the "p" (print) command like so:

  (gdb) p strs
  $3 = (char **) 0x7fffffffdd08

- Since strs is a pointer, we can dereference it with any of the normal c symbols (*, ->, [n]) and do pointer arithmetic on it

  (gdb) p *strs
  $6 = 0x148a04289b940000 <error: Cannot access memory at address 0x148a04289b940000>

- There's the bug! Lets try to see what is around it in case we have an off by one error of some kind:

  (gdb) p *(strs – 1)   // Print one pointer back. We could have also used "p strs[1]".
  $7 = 0x400a44 "butterfly"
  (gdb) p *(strs + 1)   // Print one pointer forward. We could have also used "p strs[1]".
  $8 = 0x0

- So one pointer forward gives a NULL pointer and one back gives the string "butterfly". It looks like this element in the array is corrupted? Maybe. To find out we must go deeper

GDB Example – Source Code

- When looking at the stack trace, we determined that echo was recursive. Lets look at its definition
- To print the source code of a location you can use the "l" (list) command

```
(gdb) l echo  // List source code in the 10 lines surrounding the definition of echo
46              // broken but malformed arrays of non-null terminated arrays will break inside
47              // this)
48              //
49              // An example of a NULL terminated array of strings is:
50              // char** strs = { "Hello", "World!", NULL };
51              void echo(char** strs) {
52                if (*strs != NULL) {
53                  printf("%s ", *strs);
54                  fflush(stdout);
55
(gdb)  // Pressing enter again prints the next 10 lines. Generally, pressing enter without a command runs the last command
56                  echo(strs + 1); // Recurse on the next element in the string array
57                }
58              }
59
60              // Print a **NULL terminated** array of strings to standard out and then print the
61              // array backwards (This is broken).
62              void echoohce(char** strs) {
63                char** iter;
64                char** stop_beginning = strs - 1; // HINT: What could this possibly used for?
```

GDB Example – Source Code

- Ignoring the comment above the echo function these are the important bits:

```
51     void echo(char** strs) {
52       if (*strs != NULL) {
53         printf("%s ", *strs);
54         fflush(stdout);
56         echo(strs + 1); // Recurse on the next element in the string array
57       }
58     }
```

- The echo function will not execute if the dereferenced string is a pointer to NULL. It prints the string and then recurses on the next element in the strs array

- With this we can see that the echo function approached its current position from the lower end of the array

- We still don't know what exactly we are printing, but the previous stack frame's *strs was "butterflies" which gives us something to look for where the program broke

GDB Example

- Looking back at the backtrace on slide 9 we can see that the first echo was called from main which is stack frame #10

- Lets go to main:

  (gdb) f 10
  #10 0x00000000004008ed in main (argc=1, argv=0x7fffffffde08) at bugs.c:124
  124   echo(bug_info.sentence);

- We see that echo is being called on "bug_info.sentence". Lets look at it:

  (gdb) ptype bug_info
  type = struct Storage {
      intptr_t num_bugs_on_mars;
      const char *scary_bug;
      char *sentence[6];
      const char *colorful_bug;
      intptr_t num_bugs_on_earth;
      intptr_t num_bugs_on_venus;
      char *useless_bug;
  }

  (gdb) p bug_info
  $10 = {num_bugs_on_mars = 0, scary_bug = 0x400a57 "~~~~~~~~ SPIDER!!! ~~~~~~~~", sentence = {0x602010 "The", 0x602030 "most", 0x602050 "useless", 0x602070 "bug", 0x602090 "is", 0x6020b0 "a"}, colorful_bug = 0x400a44 "butterfly", num_bugs_on_earth = 148000000000000000000, num_bugs_on_venus = 0, useless_bug = 0x400a4e "mosquito"}

GDB Example

- The echo function was expecting a NULL terminated array and sentence is not NULL terminated

- echo kept printing past the end of the array and printed `colorful_bug` ("butterfly") then tried to dereference and print `num_bugs_on_earth` which was storing an integer, not a valid pointer

- Note the value of num_bugs_on_earth. Lets look back at the problematic echo frame again:

```
(gdb) p bug_info.num_bugs_on_earth
$11 = 1480000000000000000
(gdb) f 2
#2  0x0000000000400717 in echo (strs=0x7fffffffdd08) at bugs.c:53
53          printf("%s ", *strs);
(gdb) p *strs
$12 = 0x148a04289b940000 <error: Cannot access memory at address 0x148a04289b940000>
(gdb) p (intptr_t) *strs
$13 = 1480000000000000000
```

GDB Example

- This is a buffer overflow error
- If echo changed the pointers on each of the elements it reached we would have unintentionally lost the "butterfly" string and changed the "num_bugs_on_earth" variable and stopped at "num_bugs_on_venus" (with it's value of 0 or NULL)
- There would have not been a segmentation fault and it could be more difficult to track down the problem
- Lets eliminate this bug
- Change the length of the sentence array in the Storage structure to 7 and add add bug_info.sentence[6] = NULL; to line 104

GDB Example 2

- Note: Example 2 assumes you have fixed the first problem
- There is another segmentation fault but lets first look at the line:

  `` `0` � `p` � `� `a is bug colorful most The ~~~~~~~~ SPIDER!!! ~~~~~~~~ ``

- The first part will be gibberish and it will vary between program runs
- Also, if we look at the expected output in the file bugs_desired.txt we see that the "~~~~~~~~ SPIDER!!! ~~~~~~~~" string is not present
- Lets find and fix the gibberish first and the spider string can be left for you to debug

GDB Example 2 - Breakpoints

- We will have to find the location in the code where this bug is without the segmentation fault

- Restart GDB after fixing the problem in Example 1 and rebuilding bugs.c

- Set a breakpoint at main with "b main" (break main):
  ```
  (gdb) b main
  Breakpoint 1 at 0x4007ed: file bugs.c, line 89.
  ```

- We can list all of the active breakpoints with "info breakpoints"
  ```
  (gdb) info breakpoints
  Num     Type           Disp Enb Address            What
  1       breakpoint     keep y   0x00000000004007ed in main at bugs.c:89
  ```

- Now run the program with the "r" command

- The program is now stopped at the beginning of main

GDB Example 2 - Watchpoints

- Now we need to think of a strategy to discover the bug. It looks like the sentence structure is being read backwards from the back to the front. It would be nice if we could set a breakpoint where it does the printing to narrow the search for the error

- Lets set a watchpoint that triggers on a memory read "rwatch" on the first element of the sentence array in bug_info:

  (gdb) rwatch bug_info.sentence[0]
  Hardware read watchpoint 2: bug_info.sentence[0]

- There are other watchpoint types as well: "watch" (value change) and "awatch" (read or write)

- We are ready to do our search

GDB Example 2 - Continue

- Now we can continue the program's execution with "c". This tells the program to run until the next breakpoint, watchpoint, or signal

```
(gdb) c
Continuing.
The current bug population of Earth is about: 1480000000000000000
The current bug population of Mars is about: 1
The current bug population of Venus is about: 0
The total bug population of the solar system is: 1480000000000000000
Hardware read watchpoint 2: bug_info.sentence[0]

Value = 0x602010 "The"
0x00000000004006f9 in echo (strs=0x7fffffffdcc0) at bugs.c:52
52        if (*strs != NULL) {
```

- It looks like the watchpoint triggered at echo which prints the sentence array

GDB Example 2 - Continue

- If we keep using this watchpoint we will be pressing enter quite a bit repeating the continue command until we see the line print

- Lets employ a search pattern to speed up our locating the problematic code. If we place a number after the "c" as in "c 2" then it will stop on the 2nd breakpoint or watchpoint to be triggered

- Our search pattern will be to double this number each time we don't see the problematic line

```
(gdb) c 2
Will ignore next crossing of breakpoint 2.  Continuing.
The most useless bug is a mosquito
Hardware read watchpoint 2: bug_info.sentence[0]

Value = 0x602010 "The"
0x00000000004006f9 in echo (strs=0x7fffffffdcc0) at bugs.c:52
52          if (*strs != NULL) {
```

GDB Example 2 - Continue

- It didn't print again so lets double it to 4

  (gdb) c 4
  Will ignore next 3 crossings of breakpoint 2. Continuing.
  The most colorful bug is a butterfly
  Hardware read watchpoint 2: bug_info.sentence[0]

  Value = 0x602010 "The"
  0x00007ffff7a6737c in vfprintf () from /lib64/libc.so.6

- Still haven't found it so double it again to 8

  (gdb) c 8
  Will ignore next 7 crossings of breakpoint 2. Continuing.
  `0`�`p`�`�` Hardware read watchpoint 2: bug_info.sentence[0]

  Value = 0x602010 "The"
  0x00000000004007bd in echoohce (strs=0x7fffffffdcc0) at bugs.c:78
  78   for (; *iter != NULL; --iter)

- We found it finally. It looks like we are in the echoohce. Also the rest of the backwards sentence structure hasn't printed out meaning we are very close to the code printing the gibberish

GDB Example 2 - Continue

- Lets check to see where echoohce is in the stack trace:

  ```
  (gdb) bt
  #0  0x00000000004007bd in echoohce (strs=0x7fffffffdcc0) at bugs.c:78
  #1  0x000000000040093b in main (argc=1, argv=0x7fffffffde08) at bugs.c:136
  ```

- For me, it looks like it is on line 136 of bugs.c. Use what ever line number you have in the highlighted region. Lets set a breakpoint there with "b bugs.c:136":

  ```
  (gdb) b bugs.c:136
  Breakpoint 3 at 0x40092b: file bugs.c, line 136.
  ```

- Now lets remove the watchpoint and the initial breakpoint at main with the "d #" (delete Num) command:

  ```
  (gdb) info breakpoints
  Num     Type           Disp Enb Address            What
  1       breakpoint     keep y   0x00000000004007ed in main at bugs.c:89
  breakpoint already hit 1 time
  2       read watchpoint keep y                      bug_info.sentence[0]
  breakpoint already hit 15 times
  3       breakpoint     keep y   0x000000000040092b in main at bugs.c:136
  (gdb) d 2
  (gdb) d 1
  ```

GDB Example 2 - Next

- Rerun the program with "r"

- It should break on the line you specified the breakpoint at

  (gdb) r
  ...
  The most useless bug is a mosquito
  The most colorful bug is a butterfly

  Breakpoint 3, main (argc=1, argv=0x7fffffffde08) at bugs.c:136
  136   echoohce(bug_info.sentence);

- Lets use "n" (next) to move onto the next instruction in the main function to double check that this is in fact the location of the error

  (gdb) n
  `0`� `p`� `�`a is bug colorful most The ~~~~~~~~ SPIDER!!! ~~~~~~~~
  139   free(bug_info.sentence[0]);

- We can see that GDB has completely executed echoohce and is back at the next instruction in the main function, and is also responsible for the spider string appearing

GDB Example 2 - Step

- Now that we are sure the violating function is echoohce, lets look inside it

- Restart the program again with the "r" command. We can use the "s" (step) command to step inside of a function:

(gdb) r
...
The most useless bug is a mosquito
The most colorful bug is a butterfly

Breakpoint 3, main (argc=1, argv=0x7fffffffde08) at bugs.c:136
136   echoohce(bug_info.sentence);
(gdb) s
echoohce (strs=0x7fffffffdcc0) at bugs.c:64
64   char** stop_beginning = strs - 1; // HINT: What could this possibly used for?

GDB Example 2 - Call

- Now use the "n" (next) command to go forward one instruction:

  ```
  (gdb) n
  67        for (iter = strs; *iter != NULL; ++iter)
  ```

- Press enter two more times:

  ```
  (gdb)
  68          printf("%s ", iter);
  (gdb)
  67        for (iter = strs; *iter != NULL; ++iter)
  ```

- Notice the call to printf didn't print anything. Printf is buffered and won't actually print until the buffer is full or a newline is reached in the output

- Lets flush the buffer. We can call any function in our program from GDB with the "call" command. So lets use "call fflush(stdout)"

  ```
  (gdb) call fflush(stdout)
  ` $1 = 0
  ```

GDB Example 2

- Lets check the arguments to printf like we did in Example 1

- Remember that printf wants a char* to place into the %s format token

```
(gdb) ptype iter
type = char **
(gdb) p iter
$2 = (char **) 0x7fffffffdcc0
(gdb) p *iter
$3 = 0x602010 "The"
```

- The problem is that iter is not dereferenced. If we dereference it we get the correct string

- If we fix the problem and rerun the program we can see that the line is now almost correct:

```
The most colorful bug is a a is bug colorful most The ~~~~~~~~ SPIDER!!! ~~~~~~~~
```