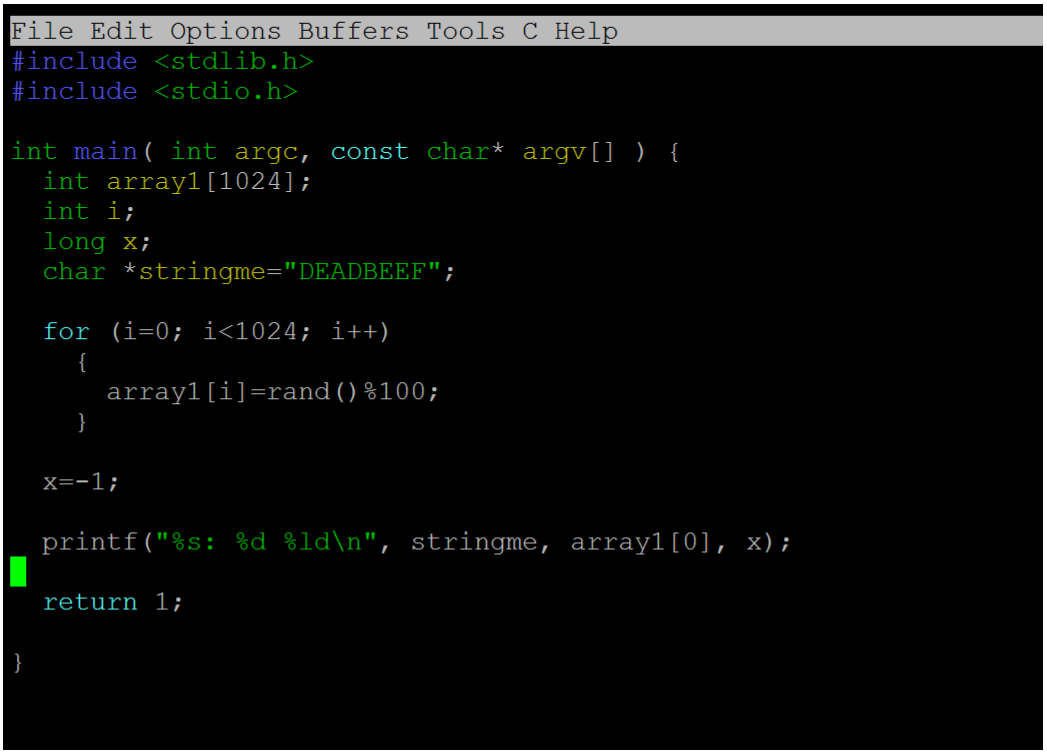
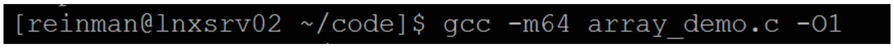
Here’s a C function:



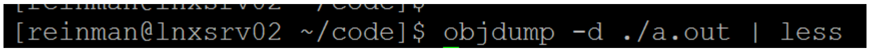
```c
#include <stdlib.h>
#include <stdio.h>

int main( int argc, const char* argv[] ) {
  int array1[1024];
  int i;
  long x;
  char *stringme="DEADBEEF";

  for (i=0; i<1024; i++)
    {
      array1[i]=rand()%100;
    }

  x=-1;

  printf("%s: %d %ld\n", stringme, array1[0], x);

  return 1;

}
```

I’ll compile it with gcc:



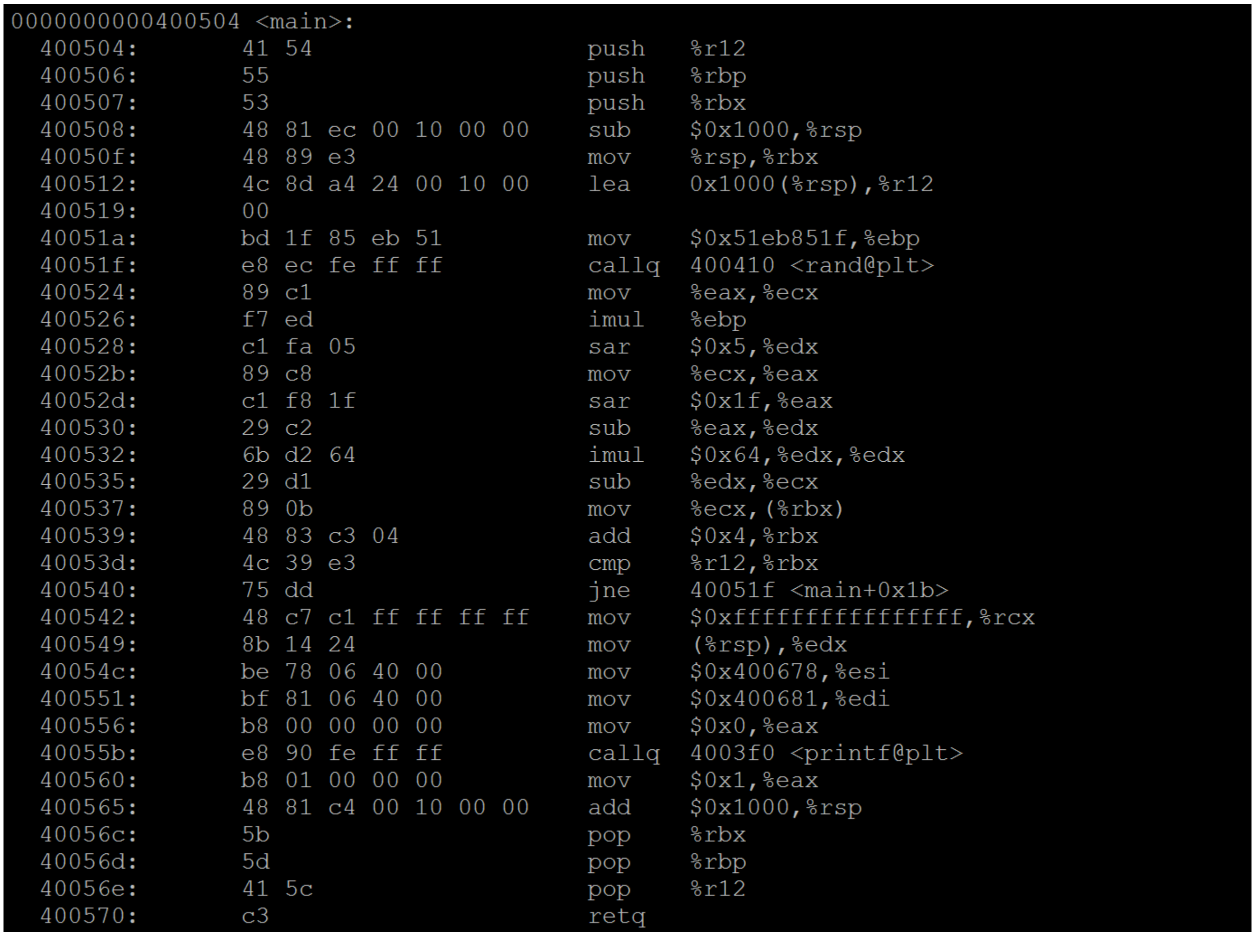
```
[reinman@lnxsrv02 ~/code]$ gcc -m64 array_demo.c -O1
```

Then use objdump:



```
[reinman@lnxsrv02 ~/code]$ objdump -d ./a.out | less
```

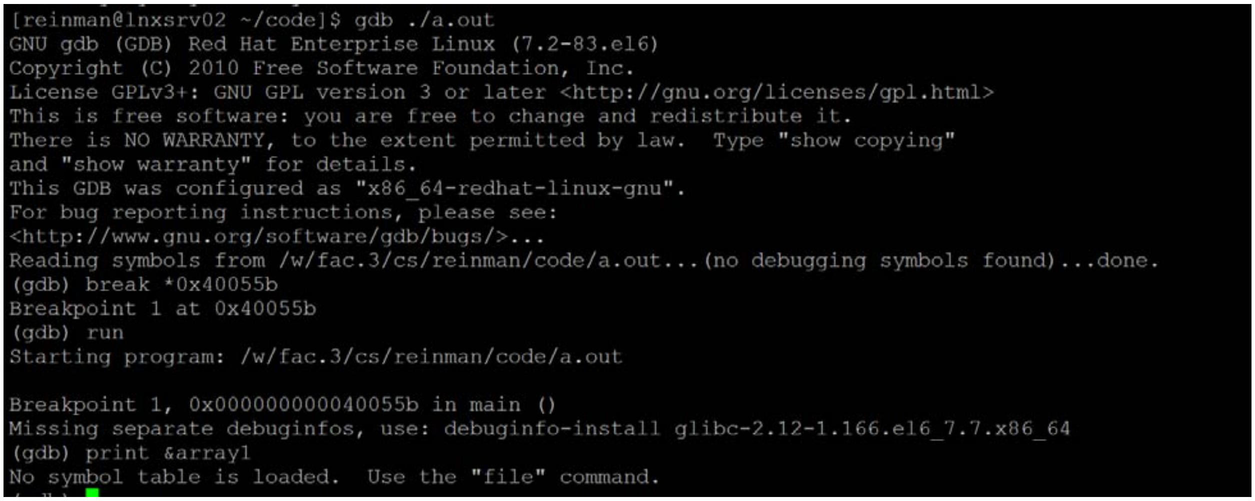
And focus in on function main:



```
0000000000400504 <main>:
  400504:       41 54                   push   %r12
  400506:       55                      push   %rbp
  400507:       53                      push   %rbx
  400508:       48 81 ec 00 10 00 00    sub    $0x1000,%rsp
  40050f:       48 89 e3                mov    %rsp,%rbx
  400512:       4c 8d a4 24 00 10 00    lea    0x1000(%rsp),%r12
  400519:       00
  40051a:       bd 1f 85 eb 51          mov    $0x51eb851f,%ebp
  40051f:       e8 ec fe ff ff          callq  400410 <rand@plt>
  400524:       89 c1                   mov    %eax,%ecx
  400526:       f7 ed                   imul   %ebp
  400528:       c1 fa 05                sar    $0x5,%edx
  40052b:       89 c8                   mov    %ecx,%eax
  40052d:       c1 f8 1f                sar    $0x1f,%eax
  400530:       29 c2                   sub    %eax,%edx
  400532:       6b d2 64                imul   $0x64,%edx,%edx
  400535:       29 d1                   sub    %edx,%ecx
  400537:       89 0b                   mov    %ecx,(%rbx)
  400539:       48 83 c3 04             add    $0x4,%rbx
  40053d:       4c 39 e3                cmp    %r12,%rbx
  400540:       75 dd                   jne    40051f <main+0x1b>
  400542:       48 c7 c1 ff ff ff ff    mov    $0xffffffffffffffff,%rcx
  400549:       8b 14 24                mov    (%rsp),%edx
  40054c:       be 78 06 40 00          mov    $0x400678,%esi
  400551:       bf 81 06 40 00          mov    $0x400681,%edi
  400556:       b8 00 00 00 00          mov    $0x0,%eax
  40055b:       e8 90 fe ff ff          callq  4003f0 <printf@plt>
  400560:       b8 01 00 00 00          mov    $0x1,%eax
  400565:       48 81 c4 00 10 00 00    add    $0x1000,%rsp
  40056c:       5b                      pop    %rbx
  40056d:       5d                      pop    %rbp
  40056e:       41 5c                   pop    %r12
  400570:       c3                      retq
```

Note that most of my code used 32 bit integers – so lots of %eax, %edx, and other 32-bit register designators. Instruction address 0x400542 uses a 64 bit register for that long x variable I used.

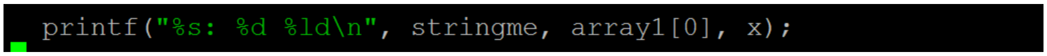
Now suppose that I want to view the contents of the array without changing the binary. Let's use gdb:



```
[reinman@lnxsrv02 ~/code]$ gdb ./a.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-83.el6)
Copyright (C) 2010 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-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /w/fac.3/cs/reinman/code/a.out...(no debugging symbols found)...done.
(gdb) break *0x40055b
Breakpoint 1 at 0x40055b
(gdb) run
Starting program: /w/fac.3/cs/reinman/code/a.out

Breakpoint 1, 0x000000000040055b in main ()
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.166.el6_7.7.x86_64
(gdb) print &array1
No symbol table is loaded.  Use the "file" command.
```

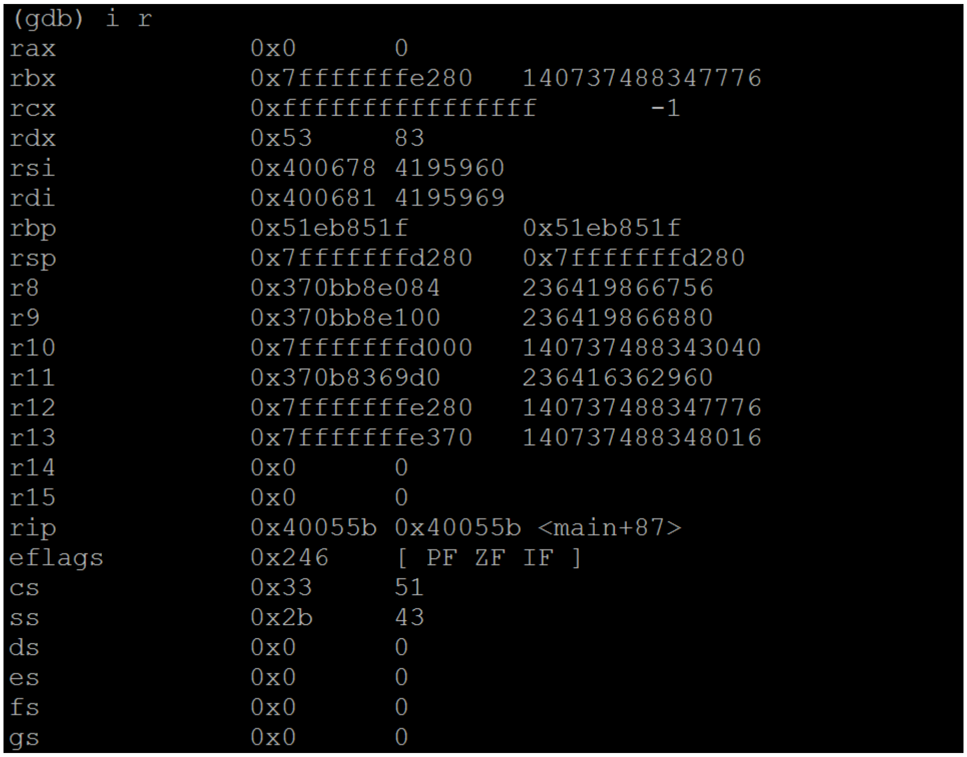
I set a breakpoint at the call to printf and ran the program. The program stopped at the breakpoint – and I tried to get the address of array1. But I compiled with –O1, so the symbol table was not kept with the binary (-Og would keep the symbol table).

Instead, let’s try to figure it out from the code. The call to printf takes four parameters:



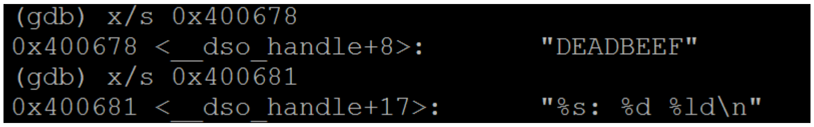
```
  printf("%s: %d %ld\n", stringme, array1[0], x);
```

Registers rdi, rsi, and rdx will hold the first three parameters (we will discuss this fact during Monday’s lecture). Let’s dump the register content using the “i r” commands:



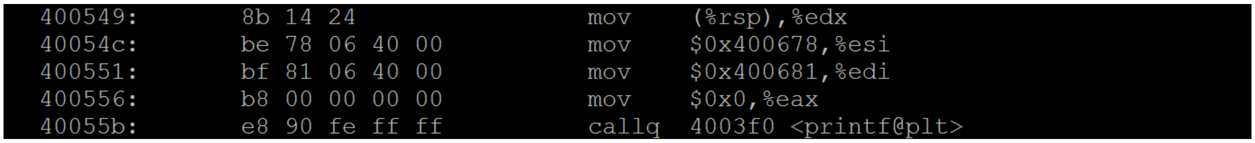
```
(gdb) i r
rax            0x0      0
rbx            0x7fffffffe280   140737488347776
rcx            0xffffffffffffffff       -1
rdx            0x53     83
rsi            0x400678 4195960
rdi            0x400681 4195969
rbp            0x51eb851f       0x51eb851f
rsp            0x7fffffffd280   0x7fffffffd280
r8             0x370bb8e084     236419866756
r9             0x370bb8e100     236419866880
r10            0x7fffffffd000   140737488343040
r11            0x370b8369d0     236416362960
r12            0x7fffffffe280   140737488347776
r13            0x7fffffffe370   140737488348016
r14            0x0      0
r15            0x0      0
rip            0x40055b 0x40055b <main+87>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

If we examine registers rsi and rdi we will see the first two parameters of the call to printf – so using the “x” command in gdb, and formatting as a string via “x/s”, we can dump the strings starting at the two addresses specified in registers rsi and rdi:



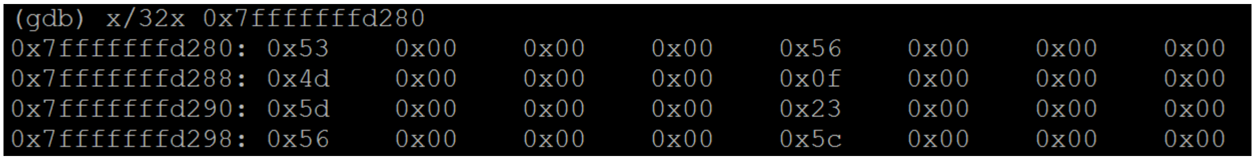
```
(gdb) x/s 0x400678
0x400678 <__dso_handle+8>:       "DEADBEEF"
(gdb) x/s 0x400681
0x400681 <__dso_handle+17>:      "%s: %d %ld\n"
```

This matches the two strings in the C code. Register rdx holds the value 0x53, which is the value in array1[0], as the printf function passes that as the third parameter. But if we look at the code, we can see the base address of array1:



```
  400549:       8b 14 24                mov    (%rsp),%edx
  40054c:       be 78 06 40 00          mov    $0x400678,%esi
  400551:       bf 81 06 40 00          mov    $0x400681,%edi
  400556:       b8 00 00 00 00          mov    $0x0,%eax
  40055b:       e8 90 fe ff ff          callq  4003f0 <printf@plt>
```

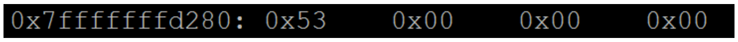
Register rdx (edx) is set by grabbing the contents pointed to by the stack pointer (rsp) – so the base address of array1 must be currently pointed to by the stack pointer (you can reconstruct this by looking at the earlier code as well). If I examine the first 32 bytes at that address (formatting them in hex):



```
(gdb) x/32x 0x7fffffffd280
0x7fffffffd280: 0x53    0x00    0x00    0x00    0x56    0x00    0x00    0x00
0x7fffffffd288: 0x4d    0x00    0x00    0x00    0x0f    0x00    0x00    0x00
0x7fffffffd290: 0x5d    0x00    0x00    0x00    0x23    0x00    0x00    0x00
0x7fffffffd298: 0x56    0x00    0x00    0x00    0x5c    0x00    0x00    0x00
```

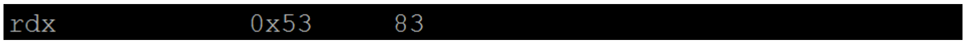
I can see the contents of the array corresponding to the first 8 elements of the array. Each array element is 4 bytes (array of int) and there are 32 bytes here – so 8 elements of the array.

The first element of the array would be the first four bytes:



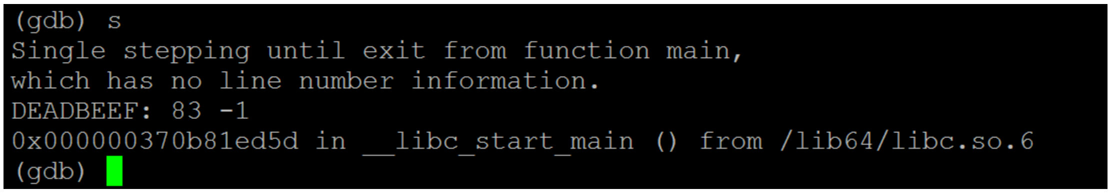
```
0x7fffffffd280: 0x53    0x00    0x00    0x00
```

Remember that x86 is little endian – so this would actually be the value 0x00000053. Which is exactly what was in register rdx from our "i r" command:



```
rdx            0x53     83
```

We could reconstruct the entire array by dumping a larger amount of memory using the "x" command. Let's finish running the program and ensure it prints what we think it should print:



```
(gdb) s
Single stepping until exit from function main,
which has no line number information.
DEADBEEF: 83 -1
0x000000370b81ed5d in __libc_start_main () from /lib64/libc.so.6
(gdb) 
```

It dumped decimal 83, which is 0x53 in hex.