Module 3
Application Debugging

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

binutils

- binutils are a collection of tools that we can use to work with object files, program binaries and libraries.
- The most commonly used binutil tools are mentioned below.
    - readelf
    - objdump
    - objcopy
    - nm
    - addr2line

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

readelf

- readelf is a command-line utility that displays information about the contents of binary files, such as object files, shared libraries, and executables.
- readelf is typically used to examine the contents and organization of binary files (specifically ELF files).
- It can display information such as the sections and segments in the file, the symbols defined in the file, the dynamic relocations and dependencies of the file, program header information and version information.
- Please refer to this comprehensive explanation of how a process and its associated resources are mapped into memory.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

readelf example

```
readelf -l /bin/ls
Elf file type is DYN (Position-Independent Executable file)
Entry point 0x6ab0
There are 13 program headers, starting at offset 64
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000003428 0x0000000000003428  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x0000000000013146 0x0000000000013146  R E    0x1000
  LOAD           0x0000000000018000 0x0000000000018000 0x0000000000018000
                 0x0000000000007458 0x0000000000007458  R      0x1000
  LOAD           0x0000000000020000 0x0000000000021000 0x0000000000021000
                 0x0000000000001278 0x0000000000002540  RW     0x1000
  DYNAMIC        0x0000000000020a98 0x0000000000021a98 0x0000000000021a98
                 0x00000000000001c0 0x00000000000001c0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x000000000001cdcc 0x000000000001cdcc 0x000000000001cdcc
                 0x000000000000056c 0x000000000000056c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000020000 0x0000000000021000 0x0000000000021000
                 0x0000000000001000 0x0000000000001000  R      0x1
...
```

Note: Position Independent Executable Code is an executable binary file format that is designed to be loaded at any memory address without modification. This property is particularly useful for security features such as Address Space Layout Randomization (ASLR).

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

objdump

- objdump provides detailed information about content and structure of object files, executable file and shared libraries.
  - Displays disassembled machine code instructions, assembly instructions, addresses and opcode.
  - Displays symbol table (functions, variables).
  - Shows the relocation and linking information.

```
manas@sandbox:~/work$ objdump -f memory
memory:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000001080
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

nm and addr2line

- nm is used to list the symbols from object files, executables, and shared libraries and their associated memory addresses.

```
manas@sandbox:~/work$ nm memory | grep first_function
0000000000001292 T first_function
```

- addr2line translates addresses into file names and line numbers in source code files.

```
manas@sandbox:~/work$ addr2line -f -e memory 0000000000001292
first_function
/home/manas/work/memory.c:30
```

- These tools are primarily used to debug Kernel OOPS.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

ldd

- ldd is used to display the shared library dependencies of an executable or shared library

```
manas@sandbox:~/work$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffeea151000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f38f92fe000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f38f90d6000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f38f903f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f38f9363000)
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GNU Debugger (GDB)

- GDB is a GNU debugger that supports languages like Ada, C/C++, Assembly, D, Fortran, Go, and Rust.
- It is mainly used to debug:
    - a process by starting it with GDB.
    - core dumps.
- GDB utilizes the debug information present in the ELF file.
- DWARF is the debugging file format used in Linux, which allows us to embed debug information in an ELF file.
- LLVM LLDB is an alternative to GDB.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Debugging file format

```
manas@manas-sandbox:~$ gcc -g -o helloworld helloworld.c

manas@manas-sandbox:~$ objdump -h helloworld
...
 27 .debug_aranges 00000030  0000000000000000  0000000000000000  00003045  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 28 .debug_info   0000017b  0000000000000000  0000000000000000  00003075  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 29 .debug_abbrev 000000de  0000000000000000  0000000000000000  000031f0  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 30 .debug_line   0000008d  0000000000000000  0000000000000000  000032ce  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 31 .debug_str    00000165  0000000000000000  0000000000000000  0000335b  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
 32 .debug_line_str 00000038  0000000000000000  0000000000000000  000034c0  2**0
                   CONTENTS, READONLY, DEBUGGING, OCTETS
```

Debug sections are separated from the .text section in the executable, allowing a non-debug binary to run on the target system while using the same ELF for debugging tools on the host system.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB Debugging Options

- We can use the following command to check if debug symbols are present in the binary: `readelf --debug-dump=decodedline <application binary>`
- We can look at the debug sections of the ELF section with `readelf -w <application binary>`.
- Compilation options available for adding debugging symbols.
    - g0: Provides no debug information.
    - g1: Produces minimal information, enough for producing back traces, but no information on variables or line numbers.
    - g2: Default debug level (same as -g). Produces symbols and line numbers required for debugging.
    - g3: Provides extra debug information, including macro definitions.
    - ggdb3: This is like g3, but generates debug information that has been tailored specifically for the GDB debugger.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Debugging & Optimization options

In order to accurately trace source code in GDB, we should disable code optimization. Following are some recommendations:

- Using `-O0` compiler optimization flag.
- Enable only GDB compatible optimization flag `-Og`.
- Use `-g` Debug flag.
- We can annotate a specific function with the compiler attribute `__attribute__((optimize("O0")))`.
- Removing the **Static** qualifier from a function can prevent the compiler from inlining the function for the purpose of optimization.
- We can use the **volatile** keyword to ensure that a variable is not optimized out.
- We can also set these equivalents in CMake using the CMake Flags directive. Please refer to the optimization example.

```
- # Set Debug configuration flags
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0 -g -Wall")
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB command line options

- Execute program with GDB.
  `gdb <program_binary_to_debug>`
- SYSROOT: Directory containing supporting files such as header files, static libraries, shared libraries etc. In GDB sysroot should point the to the location where GDB can find debug info.
  `set sysroot <Toolchain sysroot>`
- Attach GDB to running processes using the program PID.
  `gdb -p <pid_of_program_binary_to_debug>`
- When using GDB to start a program, the program needs to be run with.
  `(gdb) run`

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB command line options

Breakpoints

- A breakpoint is a debugging feature that allows a programmer to pause the execution of a program at a specific point.
- Software breakpoint: A Trap instruction is added in the software where a break point is defined. `b <line number>` or `b <function name>`
- Temporary breakpoint: Stop once and remove the break point automatically. `tbreak <lineno>`
- Hardware breakpoint: For flash based execution, software breakpoint is not ideal. Use hardware breakpoints (CPU architecture-dependent). `hbreak <lineno>`

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB command line options

Watchpoints

- Watchpoints break the program when a selected variable's value changes.
  - watch `<symbol>`, watch the specified symbol.
  - watch -l `<address>`, example: `watch -l *(int *) 0x555555558014`.
  - info watchpoints, list current watchpoints
  - To disable a watchpoint: disable `<watchpoint_number>`.
  - To delete a watchpoint: delete `<watchpoint_number>`.
  - rwatch `<symbol>`, Stops if the address or symbol is read.
  - awatch `<symbol>`, Stops if the address or symbol is accessed (similar to watch).

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB command line options

File command

- GDB provides an option of loading the program and files through `file <filename>` command from GDB prompt.
- This can be useful in many instances, for example, to load a core dump, or a symbols file or another binary that your program interacts with.
    - Load the program using file command.
    - Provide the arguments using `set args` command.
    - Check the argument using `show args` command.
    - run it using `run` command.
- If GDB is attached to a running process using `gdb -p <PID>`, we can load the executable (with -g compiled) and symbol table using file command to debug the attached process.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB command cheat-sheet

Breakpoints

=> Put a breakpoint at the entry of function foobar() - break foobar (b)
=> Put a breakpoint in foobar.c, line 10 - break foobar.c:10
=> Conditional Breakpoint - break <lineno> if <condition is true>
=> List break point - info break
=> Delete breakpoint - delete break <NUM>
=> Delete all breakpoints - clear
=> Conditional watchpoint - watch <a/s> if <condition is true>

Listing Source Code

=> list the source code
list <LINENUM> (l)
list <FILE:LINENUM>
list <FUNCION>
list <FILE:FUNCTION>
list <*ADDRESS>

Navigation

=> Continue the execution after a breakpoint - continue (c)
=> Continue to the next line, stepping over function calls - next (n)
=> Continue to the next line, entering into subfunctions - step (s)
=> Continue to the next instructions - stepi (s)
=> call a function from GDB command line - call function(a,b)

Thread Debugging

=> Show all active threads IDs - info threads
=> Select a thread by ID - thread <N>
=> Restrict breakpoint to a particular thread - break <lineno> thread <id>
=> Show backtrace of all the threads. - thread apply all bt

Printing

=> Print the variable var, the register $reg or a more complicated reference. - print var, print $reg or print task->files[0].fd (p)
=> Print structures with all their members in a human readable format - set print pretty on - print *struct_pointer
=> print object with consecutive data - print *array@len
=> Prints memory of test as a decimal byte. - x /db &test
=> Display architecture registers - info registers (b)
=> Display all local variable in a stack frame - info local
=> Display all function symbols available and their addresses. - info functions
=> Display all variable symbols available and their addresses. - info variables

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

GDB command cheat-sheet (Continued)

Signals

=> info signal -> Provides information about the status and behaviour of signals.

=> info handle -> Allows one change the way how signal is handeled in GDB. The options are:

- nostop: do not stop the program but print the signal occured
- stop: stop program when signal occurs
- print: print a message when signal occur
- noprint: do not mention when a signal occur
- pass: Allow the program to see the signal so it can be handeled
- nopass: do not pass the signal to your program

=> handle signal keyword -> Assign a handle to a signal.

- Example: handle SIGILL nostop

=> signal SEGSEGV -> Deliver SEGV signal to current program

Micellaneous

=> Change the current stack frame - frame <N>
=> Display the program stack - backtrace (bt)
=> Disassemble a function - disassemble <function>
=> Generate the core file - generate-core-file

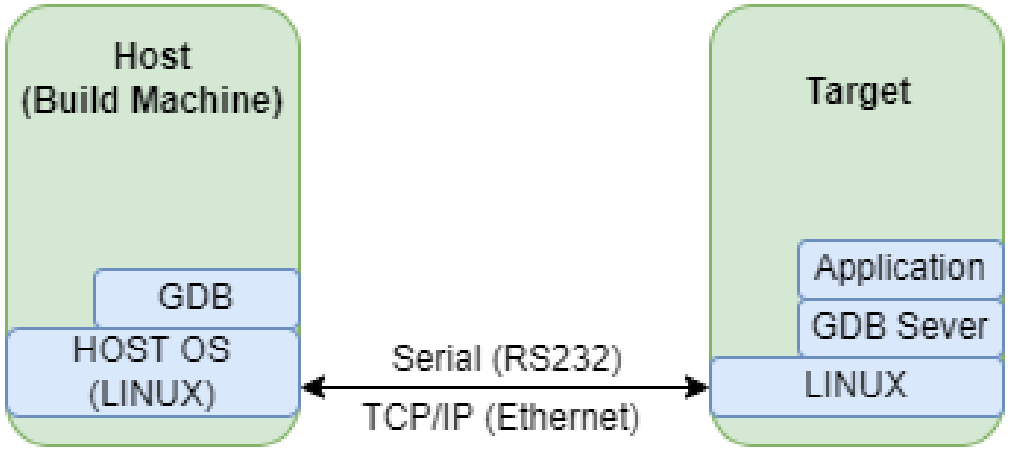
Shared Library Debugging

=> Display loaded shared library - info sharedlibrary
=> Attempt to load shared lib symbols - sharedlibrary (shared)
=> Display information about a specific shared library - info sharedlibrary <shared_library_name>
=> Display memory mappings of the entire process, including shared libraries. - info proc mappings
=> Set a prefix for locating shared library files in a specific directory. - set solib-absolute-prefix <path>
=> Set the search path for shared libraries. - set solib-search-path <path>
=> Reads additional symbol table information from the file filename - add-symbol-file <Filename> <Load Address>

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Remote debugging

- Remote debugging lets developers debug code on a remote system from their local machine.
- Used when development and target systems are physically separated (e.g., embedded or remote servers).
- Application runs under a GDB server on target is debugged using a GDB debugger on host PC over ethernet or serial port.
- DWARF specification enables debugging on an embedded target through a host PC.
- GDB server uses ptrace() to control application execution.



Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Remote debugging

Target side

- Execute the program using GDB server
  Over Network: `gdbserver localhost:[PORT] [executable] [args]`
  Over Serial: `gdbserver /dev/ttyS0 [executable] [args]`

Host side

- Execute application with GDB `gdb-multiarch -tui [executable]`
- Connect to remote gdb server

```
gdb> target extended-remote [IP]:[PORT] (Networking)
gdb> target remote /dev/ttyUSB0 (Serial)
```

- Point to shared libraries using SYSROOT `gdb> set sysroot [SYSROOT PATH]`

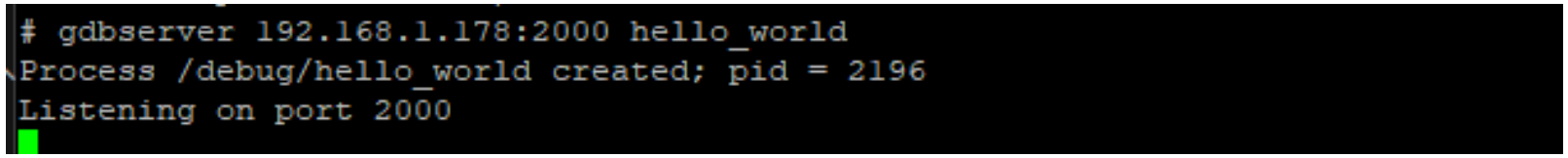
Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Remote debugging with VSCode

- Please refer to this link for instructions on configuring VSCode for debugging.

Target side

- Execute the program using GDB server on the target
  `gdbserver 192.168.1.178:2000 hello_world`

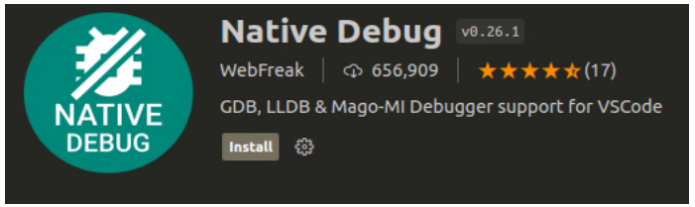


- gdbserver starts listening on port 2000 for a connection from the host side.

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Remote debugging with VSCode

Host side



- We configure VSCode by adding a debug configuration in Run->Add Configuration.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "gdb",
      "request": "attach",
      "name": "Attach to gdbserver",
      "executable": "/debug/hello_world",
      "target": "192.168.1.178:2000",
      "remote": true,
      "cwd": "${workspaceRoot}",
      "gdbpath": "/usr/bin/gdb-multiarch",
      "autorun": [
        "set sysroot /media/username/buildroot/buildroot-2023.02.8/output/staging"
      ]
    }
  ]
}
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Extending GDB with python

- Python code can be run directly inside the GDB process. This provides a powerful scripting environment for debugging.
- Standard Python libraries can be imported and these can be used to further manipulate the debug environment.

```
gdb main
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...
(gdb) python
>import os
>print(f"The PID of this process is {os.getpid()}")
>end
The PID of this process is 696028
(gdb)
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Extending GDB with python (continued)

- Python has a gdb module which is specifically designed to aid debugging. We can import this into gdb with:

```
python import gdb
```

- The execute command allows us launch the application under python control

```
gdb PythonGDBExample
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
...

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from PythonGDBExample...
(gdb) python import gdb
(gdb) python gdb.execute('start')
Temporary breakpoint 1 at 0x1171: file /home/johnos/LinuxProgramming/GeneralLinuxDebugging/Examples/PythonAndGDB/PythonGDBExample.c, line 7.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main () at /home/johnos/LinuxProgramming/GeneralLinuxDebugging/Examples/PythonAndGDB/PythonGDBExample.c:7
7           print_python_gdb_example("Example");
(gdb)
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Extending GDB with python (continued)

- The python gdb module also allows us set a breakpoint.

```
(gdb) python bp = gdb.Breakpoint('PythonGDBExample.c:12')
Breakpoint 2 at 0x555555555193: file ../Examples/PythonAndGDB/PythonGDBExample.c, line 12.
```

- We can enable or disable this breakpoint programmatically

```
(gdb) python bp.enabled = False
<gdb.Breakpoint object at 0x7fbec3f4efa0>
(gdb) python print(bp.enabled)
False
(gdb)
```

- The gdb module allows us dynamically manipulate the debug environment. Refer to the Python GDB example.

```
(gdb) python bp.enabled = True
(gdb) run
Breakpoint 2, print_python_gdb_example (message=0x555555556008 "Example") at ../Examples/PythonAndGDB/PythonGDBExample.c:12
12          printf("\nWelcome to the Use of Python in GDB %s!\n", message);
(gdb)
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Extending GDB with python (Resources)

You can find additional information and examples here:

- https://sourceware.org/gdb/current/onlinedocs/gdb.html/Python-API.html
- https://interrupt.memfault.com/blog/automate-debugging-with-gdb-python-api
- https://undo.io/resources/gdb-watchpoint/how-write-user-defined-gdb-commands-python
- https://www.lse.epita.fr/lse-winter-day-2013/slides/gdb-python.pdf

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Shared Library Debugging (1)

- We will use shared library example to demonstrate shared library debugging using GDB.
- For the purpose of debugging, the library has a explicit segfault condition.
- We can use the `LD_TRACE_LOADED_OBJECTS` environment variable to trace the dynamically loaded objects (equivalent to ldd command).

```
manas@sandbox:~/work/SharedLibGDB$ LD_TRACE_LOADED_OBJECTS=1 LD_LIBRARY_PATH=. ./main
        linux-vdso.so.1 (0x00007ffc5216f000)
        libmylib.so => ./libmylib.so (0x00007fe8f9117000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe8f8edc000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe8f911e000)
```

- As GDB points out, we are using non-debug version of application and library.

```
manas@sandbox:~/work/SharedLibGDB$ gdb main
...
(No debugging symbols found in main)
....
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Shared Library Debugging (2)

- Point the LD_LIBRARY_PATH env variable to current directory.

```
(gdb) set env LD_LIBRARY_PATH=.
```

- Set up break point at main and execute the program.

```
(gdb) b main
Breakpoint 1 at 0x1171
(gdb) r
Starting program: /home/manas/work/SharedLibGDB/main
...
Breakpoint 1, 0x0000555555555171 in main ()
```

- Display loaded shared libraries along with its mapping.

```
(gdb) info shared
From                To                  Syms Read   Shared Object Library
0x00007ffff7fc5090  0x00007ffff7fee335  Yes         /lib64/ld-linux-x86-64.so.2
0x00007ffff7fb7040  0x00007ffff7fb7113  Yes (*)     ./libmylib.so
0x00007ffff7da3700  0x00007ffff7f35abd  Yes         /lib/x86_64-linux-gnu/libc.so.6
(*): Shared library is missing debugging information.
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Shared Library Debugging (3)

- Load the debug version of shared library at the same start address where non-debug version of library is loaded.

```
(gdb) add-symbol-file /home/manas/work/SharedLibGDB/symbols/libmylib.so.debug 0x00007ffff7fb7040
add symbol table from file "/home/manas/work/SharedLibGDB/symbols/libmylib.so.debug" at
        .text_addr = 0x7ffff7fb7040
(y or n) y
Reading symbols from /home/manas/work/SharedLibGDB/symbols/libmylib.so.debug..
```

- The program crashes due to a segfault in the shared library. Because GDB has access to debugging symbols, it can point out the crash location in the code.

```
(gdb) c
Continuing.
Starting program...
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7fb710d in causeSegfault () at mylib.c:7
7         *ptr = 'A'; // This will cause a segfault
(gdb) bt
#0  0x00007ffff7fb710d in causeSegfault () at mylib.c:7
#1  0x000055555555518a in main ()
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

LD_PRELOAD

- LD_PRELOAD is an environment variable in Linux that allows you to specify shared libraries that should be loaded before all other libraries when a program starts.
- Preload libraries can override or intercept calls to specific functions in other libraries or programs.
- Useful for adding custom behavior, debugging, or profiling without modifying the original code.
- LD_PRELOAD can be used to inject debugging code into a program to trace its behavior or troubleshoot issues.

```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libSegFault.so ./<PROGRAM BINARY>
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

libsegfault

- libsegfault.so is a debugging library provided by glibc.
- It automatically dumps the backtrace and memory map when a program crashes with SEGFAULT.
- The library is activated at runtime via preload, without function overriding.
- It registers a signal handler before program execution, which prints the backtrace when a signal is delivered.
- libsegault.so looks at SEGFAULT_SIGNALS environment variable for the list of accepted signal (default SIGSEGV).
- The library is part of glibc-tools package. The location of the library varies depending on your Linux distribution.

```
SEGFAULT_SIGNALS=all LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libSegFault.so ./<PROGRAM BINARY>
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Crash Report

```
manas@sandbox:~/work/segfault$ LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libSegFault.so ./segfault
*** signal 11
Register dump:
 RAX: 0000000000000000   RBX: 0000000000000000   RCX: 000055922a34adc0
 RDX: 00007ffdc719ae98   RSI: 00007ffdc719ae88   RDI: 0000000000000001
 RBP: 00007ffdc719ad60   R8 : 00007fbd84c60f10   R9 : 00007fbd84c90040
 R10: 00007fbd84c8a908   R11: 00007fbd84ca5680   R12: 00007ffdc719ae88
 R13: 000055922a348182   R14: 000055922a34adc0   R15: 00007fbd84cc4040
 RSP: 00007ffdc719ad50
 RIP: 000055922a348161   EFLAGS: 00010206
 CS: 0033   FS: 0000   GS: 0000
 Trap: 0000000e   Error: 00000004   OldMask: 00000000   CR2: 00000000
 FPUCW: 0000037f   FPUSW: 00000000   TAG: 00000000
 RIP: 00000000   RDP: 00000000
 ST(0) 0000 0000000000000000   ST(1) 0000 0000000000000000
 ST(2) 0000 0000000000000000   ST(3) 0000 0000000000000000
 ST(4) 0000 0000000000000000   ST(5) 0000 0000000000000000
 ST(6) 0000 0000000000000000   ST(7) 0000 0000000000000000
 mxcsr: 1f80
 XMM0:  00000000000000000000000000000000 XMM1:  00000000000000000000000000000000
 XMM2:  00000000000000000000000000000000 XMM3:  00000000000000000000000000000000
 XMM4:  00000000000000000000000000000000 XMM5:  00000000000000000000000000000000
 XMM6:  00000000000000000000000000000000 XMM7:  00000000000000000000000000000000
 XMM8:  00000000000000000000000000000000 XMM9:  00000000000000000000000000000000
 XMM10: 00000000000000000000000000000000 XMM11: 00000000000000000000000000000000
 XMM12: 00000000000000000000000000000000 XMM13: 00000000000000000000000000000000
 XMM14: 00000000000000000000000000000000 XMM15: 00000000000000000000000000000000
Backtrace:
./segfault(causeSegmentationFault+0x18)[0x55922a348161]
./segfault(main+0x12)[0x55922a348194]
/lib/x86_64-linux-gnu/libc.so.6(+0x29d90)[0x7fbd84a6fd90]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0x80)[0x7fbd84a6fe40]
./segfault(_start+0x25)[0x55922a348085]
Memory map:
55922a347000-55922a348000 r--p 00000000 08:20 968310                     /home/manas/work/segfault/segfault
55922a348000-55922a349000 r-xp 00001000 08:20 968310                     /home/manas/work/segfault/segfault
55922a349000-55922a34a000 r--p 00002000 08:20 968310                     /home/manas/work/segfault/segfault
55922a34a000-55922a34b000 r--p 00002000 08:20 968310                     /home/manas/work/segfault/segfault
55922a34b000-55922a34c000 rw-p 00003000 08:20 968310                     /home/manas/work/segfault/segfault
55922c234000-55922c255000 rw-p 00000000 00:00 0                          [heap]
7fbd84a23000-7fbd84a26000 r--p 00000000 08:20 6556                       /usr/lib/x86_64-linux-gnu/libgcc_s.so.1
.....
7fbd84c61000-7fbd84c6e000 rw-p 00000000 00:00 0
7fbd84c81000-7fbd84c82000 r--p 00000000 08:20 989591                     /usr/lib/x86_64-linux-gnu/libSegFault.so
7fbd84c82000-7fbd84c85000 r-xp 00001000 08:20 989591                     /usr/lib/x86_64-linux-gnu/libSegFault.so
7fbd84c85000-7fbd84c86000 r--p 00004000 08:20 989591                     /usr/lib/x86_64-linux-gnu/libSegFault.so
7fbd84c86000-7fbd84c87000 r--p 00005000 08:20 989591                     /usr/lib/x86_64-linux-gnu/libSegFault.so
7fbd84c87000-7fbd84c88000 rw-p 00006000 08:20 989591                     /usr/lib/x86_64-linux-gnu/libSegFault.so
Segmentation fault
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Backtrace analysis

- The given backtrace is from segfault example.
- Using nm we can get the absolute address of *causeSegmentationFault()* call.

```
manas@sandbox:~/work/segfault$ nm segfault
...
0000000000001149 T causeSegmentationFault
...
```

- Adding 0x1149 + 0x18 (offset in backtrace) gives us an absolute address (0x1161) of crash location in the binary.
- Using addr2line we can point to the code which is causing the crash.

```
manas@sandbox:~/work/segfault$ addr2line -e segfault  0x1161
/home/manas/work/segfault/segfault.c:7
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Core dump

- Kernel has the ability to save a program's memory snapshot in a core file when it encounters a critical error (segmentation fault) or crashes during execution.
- Core dumps are essential for post-mortem debugging, containing memory snapshots, variable values, call stack, and CPU registers at the time of the crash.
- To enable core dumps set RLIMIT_CORE to unlimited.
  `ulimit -c unlimited`
- Naming convention of coredump file can be modified by writing to:

```
echo "/corefile/%e-%p" > /proc/sys/kernel/core_pattern

/corefile -> Directory to dump core file
%e -> Executable name
%p -> Process ID
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

Core Dump analysis with GDB

- Execute the segmentation fault example, which will generate a core dump file upon encountering a SIGSEGV signal.

```
manas@sandbox:~/work/segfault$ ./segfault
Segmentation fault (core dumped)
manas@sandbox:~/work/segfault$ ls /home/manas/work/corefile/
segfault-2771
```

- Investigate the core dump using GDB.

```
manas@sandbox:~/work/segfault$ gdb segfault -c /home/manas/work/corefile/segfault-2771

Reading symbols from segfault...
Core was generated by `./segfault'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000562441f39161 in causeSegmentationFault () at /home/manas/work/segfault/segfault.c:7
7         int value = *ptr;
(gdb) bt
#0  0x0000562441f39161 in causeSegmentationFault () at /home/manas/work/segfault/segfault.c:7
#1  0x0000562441f39194 in main () at /home/manas/work/segfault/segfault.c:15
```


Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0

References

- Debugging Embedded Devices using GDB, Chris Simmonds
  - Slides: https://elinux.org/images/0/01/Debugging-with-gdb-csimmonds-elce-2020.pdf
  - Video: https://www.youtube.com/watch?v=JGhAgd2a_Ck
- Debugging Embedded Devices Using GDB, Mike Anderson
  - https://www.youtube.com/watch?v=FnfuxDVFcWE
- Brendan Gregg's GDB tutorial: https://www.brendangregg.com/blog/2016-08-09/gdb-example-ncurses.html
- Extending GDB using python: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Python.html
- Debugging shared library
  - https://medium.com/@johnos3747/shared-libraries-in-c-programming-ab149e80be22
  - https://amir.rachum.com/shared-libraries/
- Enable coredump on Ubuntu OS
  - https://www.baeldung.com/linux/core-dumps-path-set

Linux Debug Training © 2024 John O'Sullivan | Manas Marawaha is licensed under CC BY-SA 4.0