

Dynamic Analysis and Debugging

License



This work by Thalita Vergilio at Leeds Beckett University is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

Contents

[License](#)

[Contents](#)

[Introduction](#)

[Creating a Safe Execution Environment](#)

[Basic Dynamic Analysis](#)

[Advanced Dynamic Analysis](#)

[Running a program in GDB](#)

[Main function disassembled in GDB](#)

[Breakpoints](#)

[Setting a breakpoint at the start of main\(\)](#)

[GDB showing ASM layout](#)

[GDB showing REGS layout](#)

[GDB showing list of breakpoints](#)

[GDB showing EFLAGS register](#)

[Examining Memory Locations](#)

[GDB running the simple-types program](#)

[Examining the contents of a memory address containing a single character](#)

[Examining the contents of a memory address containing an unsigned integer](#)

[Examining the contents of a memory address containing a floating-point number](#)

[Meta-CTF](#)

[General tips](#)

[GdbIntro](#)

[GdbRegs](#)

[GdbSetmem](#)

[GdbPractice](#)

[GdbParams](#)

[Conclusion](#)

Introduction

Having learned how to perform static malware analysis using Ghidra, we now turn our attention to dynamic malware analysis, which involves executing suspicious programs in a secure and controlled environment to gather information about their runtime behaviour. As we have seen, Ghidra offers very advanced features to help us reverse engineer binary code. However, one of its known weaknesses up until release 10.0.0 (June 2021) had always been the lack of a built-in debugger. In order to observe the behaviour of suspicious malware as it executes, we are going to learn how to use a new tool this week: the GNU Debugger (GDB).

Please note that, for this module, we are going to use the standard standalone version of GDB. It is worth pointing out, however, that both GDB and WinDbg have recently been integrated into Ghidra, so you may wish to experiment with them outside of this lab. Whichever way you choose to work in the future, the functionality is the same, as are the commands and shortcuts you learn in this lab.

Creating a Safe Execution Environment

Since dynamic malware analysis involves executing potentially malicious programs that could potentially harm your host system and/or network, it should only be carried out in a safe environment. For this module, we take care of this for you by providing learning challenges based on non-malicious binaries and giving you safe VMs to investigate real malware samples for your assignment.

Looking ahead into your future career as a malware analyst, however, it is important that you familiarise yourself with different ways in which you can create a secure environment for dynamic malware analysis. This is covered in this week's recommended reading, and you should aim to have at least theoretical knowledge of the options available.

Basic Dynamic Analysis

Typically, after performing initial static analysis of a suspicious executable, you would proceed to basic dynamic analysis, where you execute the program in a controlled environment while monitoring how it interacts with the host system and network to understand its runtime behaviour. The table below describes the different levels at which monitoring can be configured.

Monitoring Level	Description
Processes	Monitor the processes running in the host machine to see if any new processes are created as a result of executing the program.
File System	Observe how the malware interacts with the file system - are any new files created or existing files deleted/modified?
Registry	On Windows systems, check if new registry entries are created or existing ones modified.
Network	Monitor network traffic for activity generated by the program. Check if the program is listening on any open ports.

Advanced Dynamic Analysis

Although very informative, basic dynamic analysis involves executing the program as a black box while monitoring it externally. A more powerful approach would allow you to control every step of the program's execution while monitoring its internal state in real-time. This can be achieved by using a debugger.

You may have come across high-level graphical debuggers such as the ones which come as standard in most browsers or IDEs. They allow you to set breakpoints, step through the execution of a program line by line, and even modify dynamic values on the fly. Low-level debuggers such as GDB are pretty similar and, once you become familiar with them, you will see that the functionality available is pretty much the same. Two main differences to note are:

- instead of stepping through the source code line by line, you will step through the assembly code, instruction by instruction;
- while a lot of debuggers give you a graphical user interface, GDB is terminal-based, so it will look a little different from what you are used to.

Before we run GDB for the first time, we will change some configuration settings so our programs are disassembled using the Intel notation (GDB defaults to AT&T).

Run:

```
vi ~/.gdbinit
```

Add the following line:

```
set disassembly-flavor intel
```

Save and close the file.

Now we are ready to start working through our first example.

Create a file called `simple-if-else.c` and enter the following code:

```
int main (void) {  
    int i = 5;  
    int j = 0;  
    if (i > 3) {  
        j = i;  
    } else {  
        return 1;  
    }  
    return 0;  
}
```

This is the same program we used on Week 4.

Compile your code using `gcc`:

```
gcc -m32 simple-if-else.c -o simple-if-else
```

Now run it in `gdb`:

```
gdb ./simple-if-else
```

You should see the gdb prompt as in the image below:

```

onocentaur@p-20-156-8-a--T-saa-with-ghidra-metactf:~$ gdb ./simple-if-else
GNU gdb (Debian 8.2.1-2+b3) 8.2.1
Copyright (C) 2018 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"..
Reading symbols from ./simple-if-else...(no debugging symbols found)...done.
(gdb) █

```

Running a program in GDB

Let's ask GDB to disassemble the program's main function.

Run:

disassemble main

```

(gdb) disassemble main
Dump of assembler code for function main:
   0x00001189 <+0>:   push   ebp
   0x0000118a <+1>:   mov    ebp,esp
   0x0000118c <+3>:   sub   esp,0x10
   0x0000118f <+6>:   call  0x11c1 <_x86.get_pc_thunk.ax>
   0x00001194 <+11>:  add   eax,0x2e6c
   0x00001199 <+16>:  mov   DWORD PTR [ebp-0x4],0x5
   0x000011a0 <+23>:  mov   DWORD PTR [ebp-0x8],0x0
   0x000011a7 <+30>:  cmp   DWORD PTR [ebp-0x4],0x3
   0x000011ab <+34>:  jle   0x11ba <main+49>
   0x000011ad <+36>:  mov   eax,DWORD PTR [ebp-0x4]
   0x000011b0 <+39>:  mov   DWORD PTR [ebp-0x8],eax
   0x000011b3 <+42>:  mov   eax,0x0
   0x000011b8 <+47>:  jmp   0x11bf <main+54>
   0x000011ba <+49>:  mov   eax,0x1
   0x000011bf <+54>:  leave
   0x000011c0 <+55>:  ret
End of assembler dump.

```

Main function disassembled in GDB

Breakpoints

You can use the `break` command to set a breakpoint. You can pass it a function name, a memory address, or even an offset from the function start.

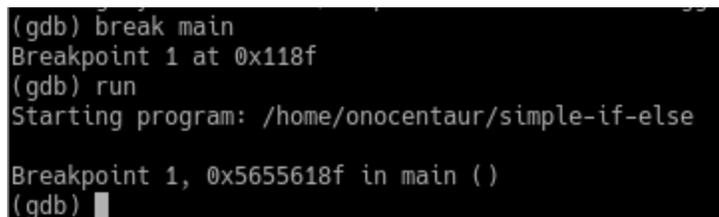
Let's set a breakpoint at the start of `main()`. You will typically start your dynamic analysis by breaking at the start of `main()` in order to watch the program's execution from the start.

Type:

```
break main
```

Now run the program:

```
run
```



```
(gdb) break main
Breakpoint 1 at 0x118f
(gdb) run
Starting program: /home/onocentaur/simple-if-else
Breakpoint 1, 0x5655618f in main ()
(gdb) █
```

Setting a breakpoint at the start of `main()`

The program executed and stopped at the breakpoint. However, the default layout does not show you a lot of information.

Change to the assembly layout:

```
layout asm
```

```

File Edit View Bookmarks Settings Help
B+> 0x5655618f <main+6> call 0x565561c1 <__x86.get_pc_thunk.ax>
0x56556194 <main+11> add eax,0x2e6c
0x56556199 <main+16> mov DWORD PTR [ebp-0x4],0x5
0x565561a0 <main+23> mov DWORD PTR [ebp-0x8],0x0
0x565561a7 <main+30> cmp DWORD PTR [ebp-0x4],0x3
0x565561ab <main+34> jle 0x565561ba <main+49>
0x565561ad <main+36> mov eax,DWORD PTR [ebp-0x4]
0x565561b0 <main+39> mov DWORD PTR [ebp-0x8],eax
0x565561b3 <main+42> mov eax,0x0
0x565561b8 <main+47> jmp 0x565561bf <main+54>
0x565561ba <main+49> mov eax,0x1
0x565561bf <main+54> leave
0x565561c0 <main+55> ret
0x565561c1 <__x86.get_pc_thunk.ax> mov eax,DWORD PTR [esp]
0x565561c4 <__x86.get_pc_thunk.ax+3> ret
0x565561c5 <__x86.get_pc_thunk.ax+4> xchg ax,ax
0x565561c7 <__x86.get_pc_thunk.ax+6> xchg ax,ax
0x565561c9 <__x86.get_pc_thunk.ax+8> xchg ax,ax
0x565561cb <__x86.get_pc_thunk.ax+10> xchg ax,ax
0x565561cd <__x86.get_pc_thunk.ax+12> xchg ax,ax
0x565561cf <__x86.get_pc_thunk.ax+14> nop
0x565561d0 <__libc_csu_init> push ebp
0x565561d1 <__libc_csu_init+1> push edi
0x565561d2 <__libc_csu_init+2> push esi
0x565561d3 <__libc_csu_init+3> push ebx
0x565561d4 <__libc_csu_init+4> call 0x56556090 <__x86.get_pc_thunk.bx>
0x565561d9 <__libc_csu_init+9> add ebx,0x2e27
0x565561df <__libc_csu_init+15> sub esp,0xc
0x565561e2 <__libc_csu_init+18> mov ebp,DWORD PTR [esp+0x28]
0x565561e6 <__libc_csu_init+22> call 0x56556000 <init>
0x565561eb <__libc_csu_init+27> lea esi,[ebx-0x108]
0x565561f1 <__libc_csu_init+33> lea eax,[ebx-0x10c]
0x565561f7 <__libc_csu_init+39> sub esi,eax
0x565561f9 <__libc_csu_init+41> sar esi,0x2
0x565561fc <__libc_csu_init+44> je 0x5655621d <__libc_csu_init+77>
native process 2189 In: main
(gdb)

```

GDB showing ASM layout

Note how your view is now split: the top pane shows the disassembled code, with breakpoints indicated on the left, while the bottom pane shows the GDB console.

Change to the registers layout:

layout regs

```

Register group: general
eax      0xf7faddc8      -134554168      ecx      0x207a7d3a      544898362      edx      0xffffd234
esp      0xffffd1f8      0xffffd1f8      ebp      0xffffd208      0xffffd208      esi      0xf7fac000
eip      0x5655618f      0x5655618f <main+6>  eflags   0x282           [ SF IF ]      cs       0x23
ds       0x2b             43              es       0x2b           43          fs       0x0

B+> 0x5655618f <main+6>      call 0x565561c1 <__x86.get_pc_thunk.ax>
0x56556194 <main+11>      add  eax,0x2e6c
0x56556199 <main+16>      mov  DWORD PTR [ebp-0x4],0x5
0x565561a0 <main+23>      mov  DWORD PTR [ebp-0x8],0x0
0x565561a7 <main+30>      cmp  DWORD PTR [ebp-0x4],0x3
0x565561ab <main+34>      jle  0x565561ba <main+49>
0x565561ad <main+36>      mov  eax,DWORD PTR [ebp-0x4]
0x565561b0 <main+39>      mov  DWORD PTR [ebp-0x8],eax
0x565561b3 <main+42>      mov  eax,0x0
0x565561b8 <main+47>      jmp  0x565561bf <main+54>
0x565561ba <main+49>      mov  eax,0x1
0x565561bf <main+54>      leave
0x565561c0 <main+55>      ret
0x565561c1 <__x86.get_pc_thunk.ax>  mov  eax,DWORD PTR [esp]
0x565561c4 <__x86.get_pc_thunk.ax+3>  ret
0x565561c5 <__x86.get_pc_thunk.ax+4>  xchg ax,ax
0x565561c7 <__x86.get_pc_thunk.ax+6>  xchg ax,ax

native process 2189 In: main
(gdb) layout regs
(gdb) █

```

GDB showing REGS layout

This gives you an additional pane on top where you can monitor the general-purpose registers.

If we look at the disassembled code, we can see that there is a `cmp` instruction on line `<main + 30>`, followed by a `jle` on `<main + 34>`. Let's set a breakpoint before the `jle` to check the results of the `cmp` instruction.

Type:

```
break *main+34
```

Remember: when we set a breakpoint in GDB, the program's execution stops BEFORE the line is executed.

To get a list of all the breakpoints you set, type:

info break

```
(gdb) info break
Num      Type      Disp Enb Address  What
1        breakpoint keep y  0x5655618f <main+6>
          breakpoint already hit 1 time
2        breakpoint keep y  0x565561ab <main+34>
(gdb) █
```

GDB showing list of breakpoints

Now that we have our second breakpoint in place, let's continue the program's execution.

Type:

`continue`

As a shortcut, you can simply type `c`. For other shortcuts and a cheat-sheet of the most commonly used commands, check the reference card below:

[GDB Quick Reference](#)

Alternatively (and we do recommend you do this), you can create your own.

When the execution stops at our second breakpoint, we can examine the EFLAGS register to determine whether the jump will be taken or not based on the result of the preceding `cmp` operation. Remember the jumps table we studied in Week 4? You can use that as a reference.

According to the table, a `jle` jump will be taken if `ZF=1` or `SF=1`. You can either **type:**

`info registers eflags`

or look at the EFLAGS register information on the top pane if you are using the REGS layout.

```

Register group: general
eax      0x56559000      1448448000      ecx
ebx      0x0              0                esp
esi      0xf7fac000     -134561792      edi
eflags   0x202          [ IF ]          cs
ds       0x2b            43              es
gs       0x63            99

B+ 0x5655618f <main+6>      call 0x565561c1 <__x86.get_pc_thunk.ax>
0x56556194 <main+11>      add  eax,0x2e6c
0x56556199 <main+16>      mov  DWORD PTR [ebp-0x4],0x5
0x565561a0 <main+23>      mov  DWORD PTR [ebp-0x8],0x0
0x565561a7 <main+30>      cmp  DWORD PTR [ebp-0x4],0x3
B+> 0x565561ab <main+34>      jle  0x565561ba <main+49>
0x565561ad <main+36>      mov  eax,DWORD PTR [ebp-0x4]
0x565561b0 <main+39>      mov  DWORD PTR [ebp-0x8],eax
0x565561b3 <main+42>      mov  eax,0x0
0x565561b8 <main+47>      jmp  0x565561bf <main+54>
0x565561ba <main+49>      mov  eax,0x1
0x565561bf <main+54>      leave
0x565561c0 <main+55>      ret
0x565561c1 <__x86.get_pc_thunk.ax>  mov  eax,DWORD PTR [esp]
0x565561c4 <__x86.get_pc_thunk.ax+3>  ret
0x565561c5 <__x86.get_pc_thunk.ax+4>  xchg ax,ax
0x565561c7 <__x86.get_pc_thunk.ax+6>  xchg ax,ax

native process 2189 In: main
(gdb) info registers eflags
eflags      0x202          [ IF ]
(gdb)

```

GDB showing EFLAGS register

As we can see, only the Interrupt Enabled (IF) flag is set. This is to be expected, since we are debugging. We can deduce that the jump was not taken. Let's check if we were correct.

Advance the execution by stepping exactly one instruction.

Type:

si

Confirm that the jump was indeed not taken

You will get plenty of practice with breakpoints in this week's CTF exercises. Don't forget to take notes and keep a table/list of helpful commands at hand, at least until you've memorised them.

Examining Memory Locations

Another very useful feature of GDB (and one which you will be using a lot) is being able to examine memory locations to read the data stored in them.

If you are still running the previous program in GDB, stop it by **typing:**

quit

Create a file called `simple-types.c` and enter the following code:

```
#include <stdio.h>

int main(void) {
    char c = 'a';
    int i = 5;
    float f = 1.5f;
    char *name = "Rampage";
    printf("%s\n", name);
    return 0;
}
```

Compile your code using `gcc`:

```
gcc -m32 simple-types.c -o simple-types
```

Now run it in `gdb`:

```
gdb ./simple-types
```

Add a breakpoint to the start of `main()`.

Change the layout to `ASM`.

Run the program.

Note a call to the `puts()` function. This is your `printf()` call in the original C code.

Add a breakpoint on that line and continue the execution so the program stops there.

```

0x56556199 <main>          lea    ecx,[esp+0x4]
0x5655619d <main+4>         and    esp,0xffffffff
0x565561a0 <main+7>          push  DWORD PTR [ecx-0x4]
0x565561a3 <main+10>         push  ebp
0x565561a4 <main+11>        mov    ebp,esp
0x565561a6 <main+13>        push  ebx
0x565561a7 <main+14>        push  ecx
0x565561a8 <main+15>        sub    esp,0x10
0x565561ab <main+18>        call  0x565561f1 <__x86.get_pc_thunk.ax>
0x565561b0 <main+23>        add    eax,0x2e50
0x565561b5 <main+28>        mov    BYTE PTR [ebp-0x9],0x61
0x565561b9 <main+32>        mov    DWORD PTR [ebp-0x10],0x5
0x565561c0 <main+39>        fld   DWORD PTR [eax-0x1ff0]
0x565561c6 <main+45>        fstp  DWORD PTR [ebp-0x14]
0x565561c9 <main+48>        lea   edx,[eax-0x1ff8]
0x565561cf <main+54>        mov   DWORD PTR [ebp-0x18],edx
0x565561d2 <main+57>        sub   esp,0xc
0x565561d5 <main+60>        push  DWORD PTR [ebp-0x18]
0x565561d8 <main+63>        mov   ebx,eax
0x565561da <main+65>        call  0x56556030 <puts@plt>
0x565561df <main+70>        add   esp,0x10
0x565561e2 <main+73>        mov   eax,0x0
0x565561e7 <main+78>        lea   esp,[ebp-0x8]
0x565561ea <main+81>        pop   ecx
0x565561eb <main+82>        pop   ebx
0x565561ec <main+83>        pop   ebp
0x565561ed <main+84>        lea   esp,[ecx-0x4]
0x565561f0 <main+87>        ret
0x565561f1 <__x86.get_pc_thunk.ax> mov   eax,DWORD PTR [esp]
0x565561f4 <__x86.get_pc_thunk.ax+3> ret
0x565561f5 <__x86.get_pc_thunk.ax+4> xchg  ax,ax
0x565561f7 <__x86.get_pc_thunk.ax+6> xchg  ax,ax
0x565561f9 <__x86.get_pc_thunk.ax+8> xchg  ax,ax
0x565561fb <__x86.get_pc_thunk.ax+10> xchg  ax,ax

```

```

native process 3703 In: main
(gdb) run
Starting program: /home/onocentaur/simple-types

Breakpoint 1, 0x565561a8 in main ()
(gdb) break *main+65
Breakpoint 2 at 0x565561da
(gdb) c
Continuing.

Breakpoint 2, 0x565561da in main ()
(gdb) █

```

GDB running the simple-types program

We are now going to examine the memory locations where we stored our local variables and try to read the data in them. The first local variable that appears in the code, in line <main+28>, appears to be 1 byte in size. It is referenced as an offset of EBP: `ebp-0x9`.

Type:

```
x $ebp-0x9
```

The `x` command defaults to hexadecimal the first time you run it. Thereafter, it defaults to the last data type used. The best approach to avoid confusion is to explicitly

tell the command how you would like to read your data¹.

Type:

```
x /c $ebp-0x9
```

We are now telling the x command to read the data as an ASCII character. The output makes more sense:

```
(gdb) x /x $ebp-0x9
0xffffd1ef: 0xffd21061
(gdb) x /c $ebp-0x9
0xffffd1ef: 97 'a'
(gdb) █
```

Examining the contents of a memory address containing a single character

Following the example above, let's check what is stored in our next local variable: ebp-0x10.

Type:

```
x /d $ebp-0x10
```

```
(gdb) x /d $ebp-0x10
0xffffd1e8: 5
(gdb) █
```

Examining the contents of a memory address containing an unsigned integer

The /d specifier reads the data as an unsigned decimal.

The next local variable, ebp-0x14, is of type float. For the sake of precision, we will specify the size of the data as a word (2 bytes), as well as the format.

Type:

```
x /fw $ebp-0x14
```

¹ Note that you won't always know what type of data is stored in an address, so be prepared to try a few different formats.

```
(gdb) x /fw $ebp-0x14
0xffffd1e4: 1.5
(gdb) █
```

Examining the contents of a memory address containing a floating-point number

Read the data stored in the remaining local variable using an appropriate format specifier. Hint: you will need to use `x` to read the data **as an address**, then use `x` again to read the data at the address returned **as a string**. Feel free to consult the [GDB Quick Reference](#).

Meta-CTF

Now that we have had a brief introduction to GDB, let's jump straight into this week's CTF challenges. There are 5 challenges this week, designed to teach you different skills in GDB as you progress through them. There is often more than one way in which you can achieve the same results - feel free to explore them.

General tips

For all the challenges, once you have found the password, run the program again **outside of GDB** to get your flag.

If at any point your console layout looks jumbled, with the text you type overwriting the text on the screen, hit **Ctrl + L** to fix it.

Remember to use:

run	execute the program from within GDB
si	"step into" the next assembly instruction
ni	"step over" the next assembly instruction (don't step into functions)
fin	"step out" of the function
c	continue, or "play"

GdbIntro

This is a nice guided introduction to GDB. Once you have identified the correct call to `strcmp()`, check the values pushed to the stack just before the function is called.

GdbRegs

An alternative to the instructions provided is to add a breakpoint at the end of the `retval_in_rax()` function and use the REGS layout to display the registers. Another option is to use “info registers” or “info registers” + the name of the register you are interested in. Remember that RAX is the 64-bit version of EAX.

GdbSetmem

Set a breakpoint on the line that does the `cmp`. When the code stops there, find out what values are being compared. Change one of them using the `set` command so the jump is not taken and the password is printed on the screen.

GdbPractice

Put a breakpoint at the start of `pwdFunction()`. When the execution stops, examine the value in EAX. Remember to read it as a string.

GdbParams

Once you have identified the function you are interested in, break just before the function is called and examine its arguments (the strings pushed to the stack). The password is one of them.

Conclusion

At this point you have:

- Been introduced to dynamic malware analysis and learned the importance of performing this type of analysis in a secure environment;
- Understood the difference between basic and advanced dynamic analysis;
- Learned how to use GDB to step through a program’s execution at assembly level, without having access to the source code;
- Solved practical challenges using MetaCTF and found 5 more flags!

Well done!

GDB is a powerful tool for performing dynamic analysis on malware for which the source code is not available. Now that you have completed this week’s work, you are ready to take on more advanced debugging challenges.