

# Reverse Engineering

2023/5/7, Sharkcode

# Sharkkcode

- ❖ CCU IM → NTHU IS
- ❖ CCU ISC
- ❖ Interested in Reverse, PWN
- ❖ BLOG :
  - ❖ <https://sharkkcode.github.io/>

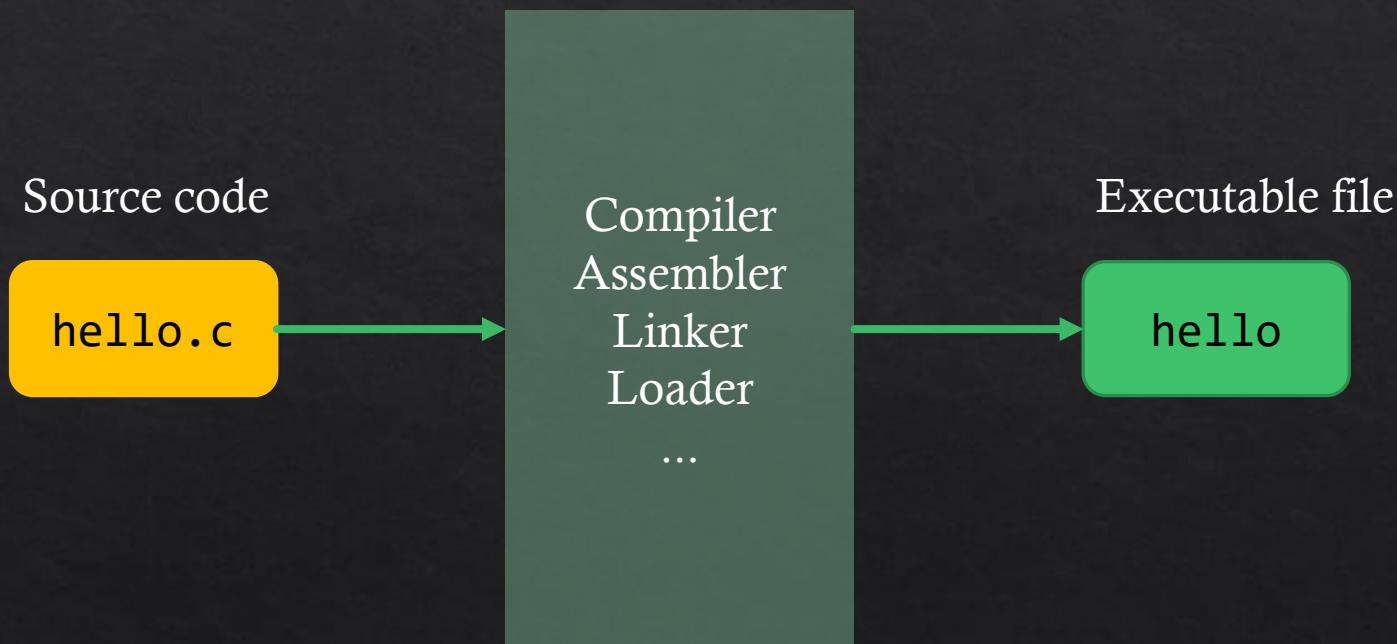


# Outline

- ❖ Reverse Engineering Overview
- ❖ Analysis
- ❖ x64 Assembly
- ❖ Memory Layout
- ❖ Demo Learning Website
- ❖ Lab
- ❖ Tools
  - ❖ GHIDRA
  - ❖ GDB
  - ❖ pwntools
- ❖ Lab

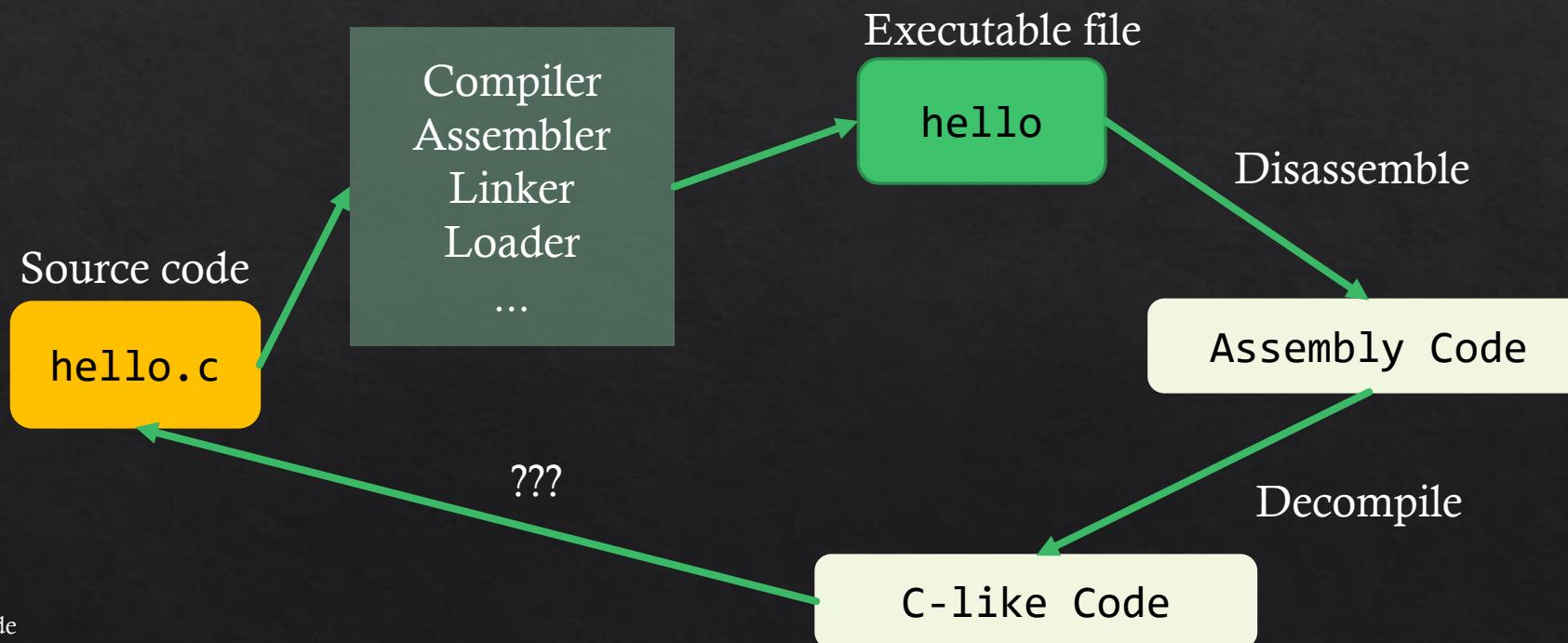
# Reverse Engineering Overview

- ❖ When we finish writing a program `hello.c`, we might want to use Compiler, Assembler, Linker, Loader to generate an executable file `hello` to execute.



# Reverse Engineering Overview

- ❖ So what exactly is Reverse Engineering doing?
  - ❖ Analyzing the files and information you have to reconstruct the behavior of a program or project, which can facilitate further actions such as writing a cracking program, etc.



# Bigger Overview?

- ❖ Figure out **what is going on** ...

通靈

# Analysis

- ❖ Statically
  - ❖ Without running...
- ❖ Dynamically
  - ❖ When running...

# x64 Assembly

- ❖ <https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf>

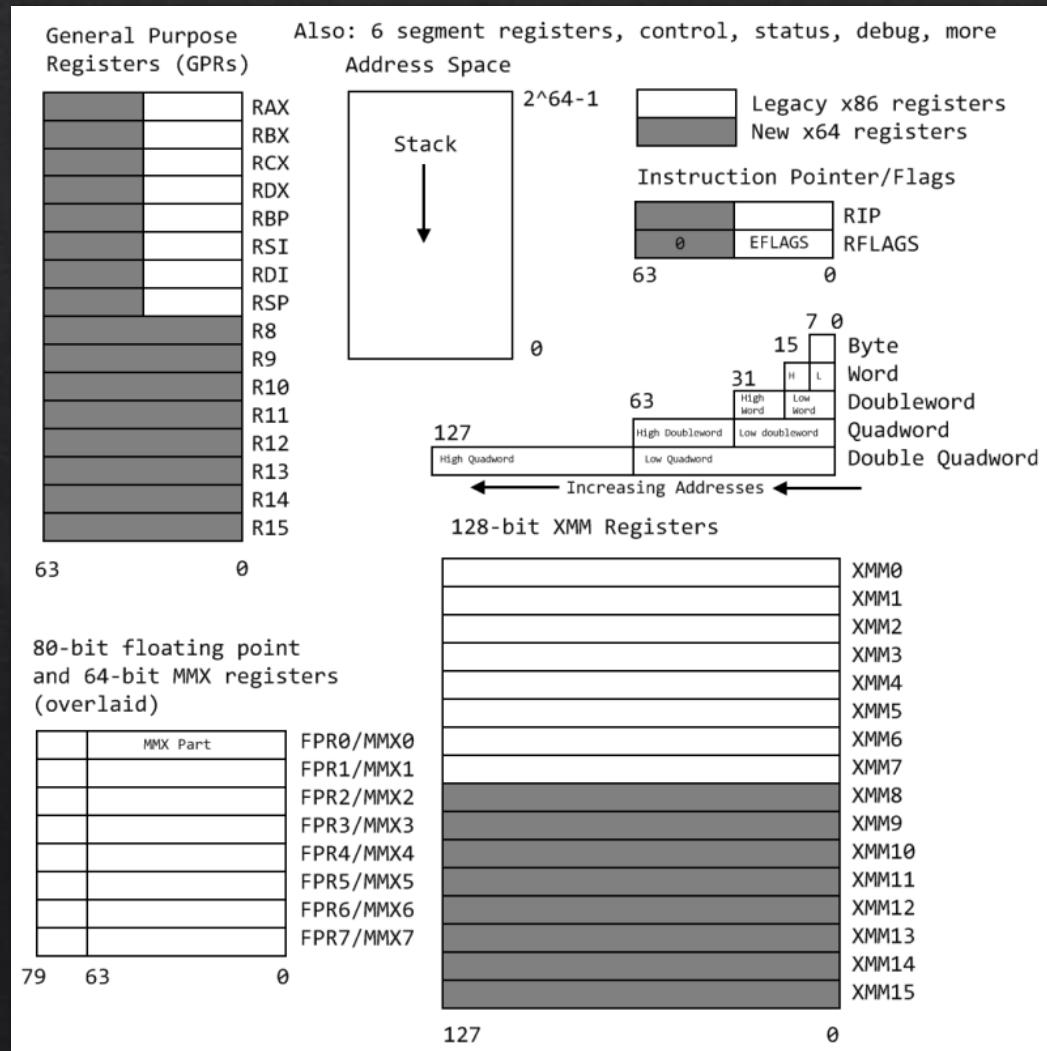


# x64 Assembly -- Cheatsheet

- ❖ [https://cs.brown.edu/courses/cs033/docs/guides/x64\\_cheatsheet.pdf](https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf)



# x64 Assembly -- General Architecture



# x64 Assembly -- Registers

## 4.3 Register Usage

There are sixteen 64-bit registers in x86-64: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rbp`, `%rsp`, and `%r8-r15`. Of these, `%rax`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rsp`, and `%r8-r11` are considered caller-save registers, meaning that they are not necessarily saved across function calls. By convention, `%rax` is used to store a function's return value, if it exists and is no more than 64 bits long. (Larger return types like structs are returned using the stack.) Registers `%rbx`, `%rbp`, and `%r12-r15` are callee-save registers, meaning that they are saved across function calls. Register `%rsp` is used as the *stack pointer*, a pointer to the topmost element in the stack.

Additionally, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.

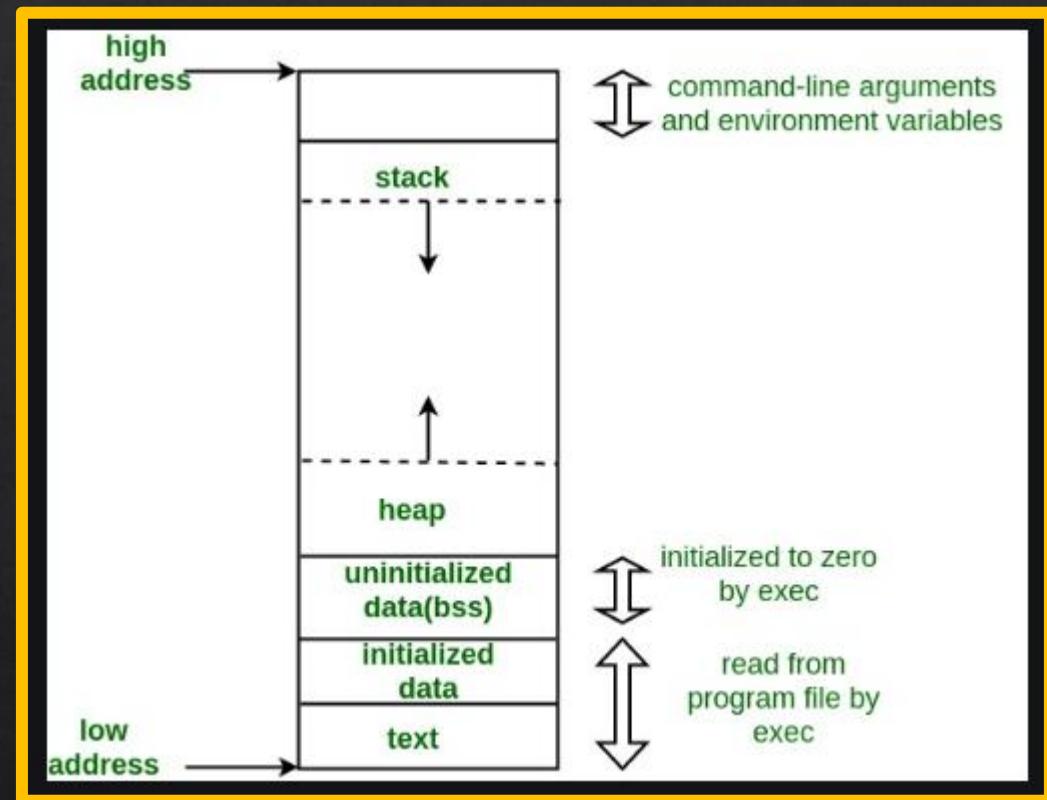
In 32-bit x86, the *base pointer* (formerly `%ebp`, now `%rbp`) was used to keep track of the base of the current stack frame, and a called function would save the base pointer of its caller prior to updating the base pointer to its own stack frame. With the advent of the 64-bit architecture, this has been mostly eliminated, save for a few special cases when the compiler cannot determine ahead of time how much stack space needs to be allocated for a particular function (see Dynamic stack allocation).

# x64 Assembly -- Instructions

- ❖ mov
- ❖ jmp
- ❖ call
- ❖ ret
- ❖ nop
- ❖ ...

# Memory Layout

- ❖ <https://www.geeksforgeeks.org/memory-layout-of-c-program/>



# Demo Learning Website

<https://godbolt.org/>

# Website Overview

The screenshot shows the Compiler Explorer interface with two tabs open:

- C++ source #1:** Contains the following C++ code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```
- x86-64 gcc 12.2 (Editor #1):** Shows the assembly output generated by the compiler:

```
1 square(int):
2         push    rbp
3         mov     rbp, rsp
4         mov     DWORD PTR [rbp-4], edi
5         mov     eax, DWORD PTR [rbp-4]
6         imul   eax, eax
7         pop    rbp
8         ret
```

At the bottom, there is an "Output" tab showing the command and execution time:

C Output (0/0) x86-64 gcc 12.2 i - 739ms (2811B) ~170 lines filtered 15

Other UI elements include a "Compiler License" link at the bottom center.

# Website Overview

The screenshot shows the Compiler Explorer interface with the following components:

- Top Bar:** Includes "Add...", "More", "Templates", "Backtrace intel Solid Sands" logo, "Share", "Policies", and "Other".
- Left Panel:** "C++ source #1" tab with code editor containing:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

A yellow callout box labeled "Choose Language" points to the "C++" dropdown.
- Middle Panel:** "x86-64 gcc 12.2" tab with assembly output:

```
1 square(int):
2
3     mov    eax, DWORD PTR [rbp-4]
4     imul   eax, eax
5     pop    rbp
6     ret
```

A yellow callout box labeled "Choose Compiler" points to the "x86-64 gcc 12.2" dropdown.
- Right Panel:** "Compiler options..." dropdown menu with a yellow callout box labeled "Choose Compiler Options".
- Bottom Status Bar:** Shows "Output (0/0) x86-64 gcc 12.2", "739ms (2811B) ~170 lines filtered", "Compiler License", and page number "(4, 2)".

# Hello World DEMO

```
// SOURCE CODE
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello World!\n");
    return 0;
}
```

# Hello World DEMO

The diagram illustrates the compilation process from C source code to assembly code. It shows two main windows: Compiler Explorer on the left and Backtrace intel on the right.

**Compiler Explorer (Left):**

- A red box highlights the C source code:

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
```

- An arrow points from the highlighted source code to the "Executable file" box in the flowchart.

**Backtrace intel (Right):**

- A red box highlights the assembly code:

```
.LC0:
    .string "Hello World!"
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], edi
    mov     QWORD PTR [rbp-16], rsi
    mov     edi, OFFSET FLAT:.LC0
    call    puts
    mov     eax, 0
    leave
    ret
```

- An arrow points from the "Executable file" box in the flowchart to the assembly code window.

**Flowchart (Bottom Left):**

```
graph TD
    A[Source code  
hello.c] --> B[Compiler  
Assembler  
Linker  
Loader  
...]
    B --> C[hello]
    C -- Executable file --> D[Assembly Code]
    D -- Disassemble --> E[c-like Code]
    E -- Decompile --> A
    A -- ?? --> E
```

The flowchart shows the following relationships:

- Source code (hello.c) feeds into Compiler Assembler Linker Loader.
- Compiler Assembler Linker Loader outputs an Executable file (hello).
- Executable file (hello) can be Disassembled into Assembly Code.
- Assembly Code can be Disassembled back into C-like Code.
- C-like Code can be Decomplied back into Source code (hello.c).
- There is a feedback loop from C-like Code back to Source code (hello.c) via a question mark (??).

Annotations:

- "which can facilitate further actions such as writing a cracking program, etc." is located above the flowchart.
- "By Sharkcode" is at the bottom left of the flowchart.
- "rikcode" is at the bottom left of the flowchart.
- "18" is at the bottom right of the image.

# Hello World DEMO

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C++ source code for a "Hello World" application. The right pane shows the generated assembly code for the x86-64 architecture using gcc 12.2.

**C++ Source Code (Left):**

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
```

**Assembly Output (Right):**

```
1 .LC0:
2     .string "Hello World!"
3
4 main:
5     push    rbp
6     mov     rbp, rsp
7     sub     rsp, 16
8     mov     DWORD PTR [rbp-4], edi
9     mov     QWORD PTR [rbp-16], rsi
10    mov     edi, OFFSET FLAT:.LC0
11    call    puts
12    mov     eax, 0
13    leave
14    ret
```

A red arrow points from the highlighted line in the C++ source code (line 5) to the corresponding assembly instruction in the right pane (line 11, `call puts`).

By Sharkcode

19

C Output (0/0) x86-64 gcc 12.2 i - 764ms (3942B) ~252 lines filtered E

(10, 1)

Compiler License

# Hello World DEMO

The screenshot shows two tabs in the Compiler Explorer interface: 'C++ source #1' and 'x86-64 gcc 12.2 (Editor #1)'. The left tab displays the C++ source code:

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
```

The right tab shows the generated assembly code:

```
1 .LC0:
2     .string "Hello World!"
3
4 main:
5     push    rbp
6     mov     rbp, rsp
7     sub     rsp, 16
8     mov     DWORD PTR [rbp-4], edi
9     mov     QWORD PTR [rbp-16], rsi
10    mov     edi, OFFSET FLAT:.LC0
11    call    puts
12    mov     eax, 0
13
14    leave
15    ret
```

A red box highlights the assembly code from line 4 to 7, which corresponds to the function prologue. A yellow callout bubble points to this box with the text 'Function Prologue'. Another red box highlights the assembly code from line 14 to 15, which corresponds to the function epilogue. A second yellow callout bubble points to this box with the text 'Function Epilogue'.

By Sharkcode

20

C Output (0/0) x86-64 gcc 12.2 i - 764ms (3942B) ~252 lines filtered E

(10, 1)

Compiler License

# Hello World DEMO

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C++ source code:

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
```

The right pane shows the generated assembly code for x86-64 gcc 12.2:

```
1 .LC0:
2     .string "Hello World!"
3
4 main:
5     push    rbp
6     mov     rbp, rsp
7     sub    rsp, 16
8     mov     DWORD PTR [rbp-4], edi
9     mov     QWORD PTR [rbp-16], rsi
10    mov    edi, OFFSET FLAT:.LC0
11    call   puts
12    mov    eax, 0
13    leave
14    ret
```

A red box highlights the instruction `sub rsp, 16`, and a yellow callout bubble points to it with the text "Stack grows".

At the bottom, the status bar shows: Output (0/0) x86-64 gcc 12.2 - 764ms (3942B) ~252 lines filtered.

Page footer: By Sharkcode (10, 1) Compiler License 21

# Hello World DEMO

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C++ source code for a "Hello World" application. The right pane shows the generated assembly code for the x86-64 architecture using gcc 12.2.

**C++ Source Code:**

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
9
10
```

**Assembly Output:**

```
1 .LC0
2
3 main:
4     push    rbp
5     mov     rbp, rsp
6     sub     rsp, 16
7     mov     DWORD PTR [rbp-4], edi
8     mov     QWORD PTR [rbp-16], rsi
9     mov     edi, OFFSET FLAT:.LC0
10    call    puts
11    mov     eax, 0
12    leave
13    ret
```

A red box highlights the function signature `int main(int argc, char **argv)` in the source code. A purple callout box in the assembly pane provides information about argument passing conventions:

Additionally, %rdi, %rsi, %rdx, %rcx, %r8, and %r9 are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.

A red box highlights the assembly instructions `mov DWORD PTR [rbp-4], edi` and `mov QWORD PTR [rbp-16], rsi`. A yellow arrow points from this box to a green callout box containing the text:

Store arguments on stack

Page footer: By Sharkcode 22

Page footer: Output (0/0) x86-64 gcc 12.2 - 764ms (3942B) ~252 lines filtered Compiler License

# Hello World DEMO

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C++ source code for a "Hello World" program:

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
```

The line `printf("Hello World!\n");` is highlighted with a red box. The right pane shows the generated assembly code for x86-64 gcc 12.2:

```
1 .LC0:
2     .string "Hello World!"
3
4 main:
5
6     mov    DWORD PTR [rbp-4], edi
7     mov    QWORD PTR [rbp-16], rsi
8     mov    edi, OFFSET FLAT:.LC0
9     call   puts
10    mov    eax, 0
11    leave
12    ret
```

A yellow box highlights the instruction `mov edi, OFFSET FLAT:.LC0`. A purple callout box provides additional context:

Additionally, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.

At the bottom, the status bar shows: Output (0/0) x86-64 gcc 12.2 - 764ms (3942B) ~252 lines filtered. The page footer includes: By Sharkcode (10, 1), Compiler License, and page number 23.

# Hello World DEMO

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays the C++ source code for a "Hello World" program, and the right pane shows the generated assembly code.

**C++ Source Code (Left Pane):**

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
9
10
```

The line `printf("Hello World!\n");` is highlighted with a red box.

**Assembly Output (Right Pane):**

```
1 .LC0:
2     .string "Hello World!"
3
4 main:
5     push    rbp
6     mov     rbp, rsp
7     sub     rsp, 16
8     mov     DWORD PTR [rbp-4], edi
9     mov     QWORD PTR [rbp-16], rsi
10    mov     edi, OFFSET FLAT:.LC0
11    call    puts
12    mov     eax, 0
13    leave
14    ret
```

The lines `mov edi, OFFSET FLAT:.LC0` and `call puts` are highlighted with a red box.

At the bottom of the interface, there is an "Output" tab showing "0/0" results, and a "Compiler License" link.

# Hello World DEMO

The screenshot shows the Compiler Explorer interface with two tabs: "C++ source #1" and "x86-64 gcc 12.2 (Editor #1)".

**C++ Source #1:**

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4
5     printf("Hello World!\n");
6
7     return 0;
8 }
```

A red box highlights the `return 0;` statement.

**x86-64 gcc 12.2 (Editor #1):**

```
8     mov    QWORD PTR [rbp-16], rsi
9     mov    edi, OFFSET FLAT:.LC0
10    call   puts
11    mov    eax, 0
12    leave
13    ret
```

A yellow box highlights the `mov eax, 0` instruction. A purple callout box provides information about x86-64 registers:

There are sixteen 64-bit registers in x86-64: %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, and %r8-r15. Of these, %rax, %rcx, %rdx, %rdi, %rsi, %rsp, and %r8-r11 are considered caller-save registers, meaning that they are not necessarily saved across function calls. By convention, %rax is used to store a function's return value if it exists and is no more than 64 bits long. (Larger return types like structs are returned using the stack.) Registers %rbx, %rbp, and %r12-r15 are callee-save registers, meaning that they are saved across function calls. Register %rsp is used as the *stack pointer*, a pointer to the topmost element in the stack.

By Sharkcode

25

Output (0/0) x86-64 gcc 12.2 - 764ms (3942B) ~252 lines filtered

(10, 1)

Compiler License

omeD gninraeL etisbeW

<https://dogbolt.org/>

# Compare with several decompilers

 Decompile Explorer

What is this?

Upload File  
Your file must be less than 2MB in size. Uploaded binaries [are retained](#).

選擇檔案 未選擇任何檔案

angr     BinaryNinja     Boomerang     dewolf     Ghidra     Hex-Rays     RecStudio     Reko     Relyze     RetDec     Snowman

Samples  
Or check out one of these samples we've provided:  
A CTF Challenge on x86 Linux

angr	BinaryNinja	Ghidra	Hex-Rays
<pre>9.2.34 1 int __init() 2 { 3     return; 4     if (false) 5     { 6         0(); 7         return; 8     } 9 } 10 long long sub_401020() 11 { 12     unsigned long long v0; // [bp-0x8] 13     v0 = 0; 14     goto *(4210576); 15 } 16 long long sub_401030() 17 { 18     int64_t var_8 = 0; 19     /* jump -&gt; nullptr */ 20 } 21 unsigned long long v0; // [bp-0x8] 22 v0 = 0; 23 return sub_401020(); 24 }</pre>	<pre>3.3.3996 (e34a955e) 1 void __init() 2 { 3     if (__gmon_start__ != 0) 4     { 5         __gmon_start__(); 6     } 7 } 8 int64_t sub_1020() 9 { 10    int64_t var_8 = 0; 11    /* jump -&gt; nullptr */ 12 } 13 int64_t sub_1030() 14 { 15    int64_t var_8 = 0; 16    /* tailcall */ 17    return sub_1020(); 18 } 19 long long sub_401040() 20 { 21     unsigned long long v0; // [bp-0x8] 22     v0 = 1; 23     return sub_401020(); 24 }</pre>	<pre>10.2.2 (9813cde2) 1 #include "out.h" 2 3 4 5 int __init(EVP_PKEY_CTX *ctx) 6 7 { 8     int iVar1; 9 10    iVar1 = __gmon_start__(); 11    return iVar1; 12 } 13 14 15 void FUN_00101020(void) 16 17 18 { 19     /* WARNING: Treating indirect jump 20     (*(code *) (undefined *) 0x0)(); */ 21     return; 22 } 23 24 25 void __cxa_finalize(void) 26 27 28 { 29     __cxa_finalize(); 30     return; 31 }</pre>	<pre>8.2.0.221215 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36</pre>

# Lab

- ❖ <https://play.picoctf.org/practice>
  - ❖ ARMassembly 1

# Tools

GDB, GHIDRA, pwntools

# Tools

## ❖ demo.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct {
5     int id;
6     char name[50];
7     float grade;
8 } Student;
9
10 int main() {
11
12     setvbuf(stdout,0,2,0);
13     setvbuf(stdin,0,2,0);
14     setvbuf(stderr,0,2,0);
15
16     int n;
17     printf("Enter the number of students: ");
18     scanf("%d", &n);
19     Student *students = (Student*)malloc(n * sizeof(Student));
20     for (int i = 0; i < n; i++) {
21         printf("Enter student %d's ID: ", i+1);
22         scanf("%d", &(students[i].id));
23         printf("Enter student %d's name: ", i+1);
24         scanf("%s", students[i].name);
25         printf("Enter student %d's grade: ", i+1);
26         scanf("%f", &(students[i].grade));
27     }
28     printf("Student information:\n");
29     for (int i = 0; i < n; i++) {
30         printf("ID: %d, Name: %s, Grade: %.2f\n", students[i].id, students[i].name, students[i].grade);
31     }
32     free(students);
33     return 0;
34 }
```

# Tools -- GHIDRA

◆ 22e

The screenshot shows the GHIDRA interface with three main panes:

- Program Trees** pane on the left, showing the file structure of the demo executable.
- Listing** pane in the center, displaying assembly code for the main function. A specific instruction at address 0010122e is highlighted with a red box: `0010122e e8 bd CALL <EXTERNAL>::__isoc99_scanf`.
- Decompiler** pane on the right, showing the decompiled C code for the main function. The highlighted line is: `__isoc99_scanf(&DAT_00102027,&local_24);`

The filter bar in the Program Trees pane is set to "main".

```
1 undefined8 main(void)
2 {
3     long in_FS_OFFSET;
4     int local_24;
5     int local_20;
6     int local_1c;
7     void *local_18;
8     long local_10;
9
10    local_10 = *(long *)(in_FS_OFFSET + 0x28);
11    printf("Enter the number of students: ");
12    __isoc99_scanf(&DAT_00102027,&local_24);
13    local_18 = malloc((long)local_24 * 0x3c);
14    for (local_20 = 0; local_20 < local_24; local_20++)
15        printf("Enter student %d's ID: ",(ulong)(local_18 +
16            __isoc99_scanf(&DAT_0010205c),(long)local_18));
17    __isoc99_scanf(&DAT_0010205c,(long)local_18);
18    printf("Enter student %d's name: ",(ulong)(local_18 +
19            __isoc99_scanf(&DAT_0010205c),(long)local_18));
20    __isoc99_scanf(&DAT_0010205c,(long)local_18);
21    printf("Enter student %d's grade: ",(ulong)(local_18 +
22            __isoc99_scanf(&DAT_0010207a));
23    }
24    puts("Student information:");
25    for (local_1c = 0; local_1c < local_24; local_1c++)
26        printf((char *) (double *) (float *) ((long)local_18 +
27            "ID: %d, Name: %s, Grade: %.2f\n",
28            (ulong) * (uint *) ((long)local_18 + (long)local_18 +
29            (long)local_18 + (long)local_1c * 0x3c));
30    }
31    free(local_18);
32    if (local_10 != *(long *) (in_FS_OFFSET + 0x28))
33        /* WARNING: Subroutine does not end with an explicit
34         * _stack_chk_fail();
35     */
36     return 0;
37 }
```

# Tools -- GDB

## ◆ 22e

```
0x000000000000200 <+1>:    mov    rax,rax
0x00005555555520e <+37>:   mov    eax,0x0
0x000055555555213 <+42>:   call   0x555555550d0 <printf@plt>
0x000055555555218 <+47>:   lea    rax,[rbp-0x1c]
0x00005555555521c <+51>:   mov    rsi,rax
0x00005555555521f <+54>:   lea    rax,[rip+0xe01]          # 0x555555556027
0x000055555555226 <+61>:   mov    rdi,rax
0x000055555555229 <+64>:   mov    eax,0x0
0x00005555555522e <+69>:   call   0x555555550f0 <__isoc99_scanf@plt>
0x000055555555233 <+74>:   mov    eax,DWORD PTR [rbp-0x1c]
0x000055555555236 <+77>:   movsxd rdx,eax
0x000055555555239 <+80>:   mov    rax,rdx
0x00005555555523c <+83>:   shl    rax,0x4
0x000055555555240 <+87>:   sub    rax,rdx
0x000055555555243 <+90>:   shl    rax,0x2
0x000055555555247 <+94>:   mov    rdi,rax
0x00005555555524a <+97>:   call   0x555555550e0 <malloc@plt>
0x00005555555524f <+102>:  mov    QWORD PTR [rbp-0x10],rax
0x000055555555253 <+106>:  mov    DWORD PTR [rbp-0x18],0x0
```

# Tools -- pwntools

## ❖ demo\_py.py

```
1 from pwn import *
2
3 io = process("./demo")
4
5 io.sendafter(b": ", b"1\n")
6
7 io.sendafter(b": ", b"2\n")
8 io.sendafter(b": ", b"3\n")
9 io.sendafter(b": ", b"4\n")
10
11 io.interactive()
12
```

# Tools -- pwntools

- ❖ `python3 demo_py.py`

```
~/rev_pwn python3 demo_py.py
[+] Starting local process './demo': pid 1741811
[*] Switching to interactive mode
[*] Process './demo' stopped with exit code 0 (pid 1741811)
Student information:
ID: 2, Name: 3, Grade: 4.00
[*] Got EOF while reading in interactive
$ █
```

# Lab

`./reverse_demo_s`

Author Sharkkcode

Your goal is to find out the right key!

# Lab

./tea

From SHELL CTF 2022

Your goal is to find out the FLAG!

# THANKS

## Q&A