

- [Reverse Lab 1](#)
  - [1. Task 1](#)
  - [2. Task 2](#)
  - [3. Task 3](#)
  - [4. Bonus](#)

# Reverse Lab 1

---

## 1. Task 1

---

gcc+llvm-mc可以正常生成目标文件:

```
→ Reverse llvm-mc-14 -filetype=obj hello_gcc.s -o hello_gcc_clang.o
```

但clang+as却不能正常生成目标文件:

```
→ Reverse as hello_clang.s -o hello_clang_gcc.o
hello_clang.s: Assembler messages:
hello_clang.s:36: Error: unknown pseudo-op: `.addrsig'
hello_clang.s:37: Error: unknown pseudo-op: `.addrsig_sym'
```

从报错信息中可以看出, 这个问题是clang中使用了as无法理解的伪代码操作引起的, 即.addrSIG和.addrSIG\_sym。根据变量名判断是与地址签名相关的内容, 用vim打开能在结尾看到以下两行代码:

```
.addrsig
.addrSIG_sym printf
```

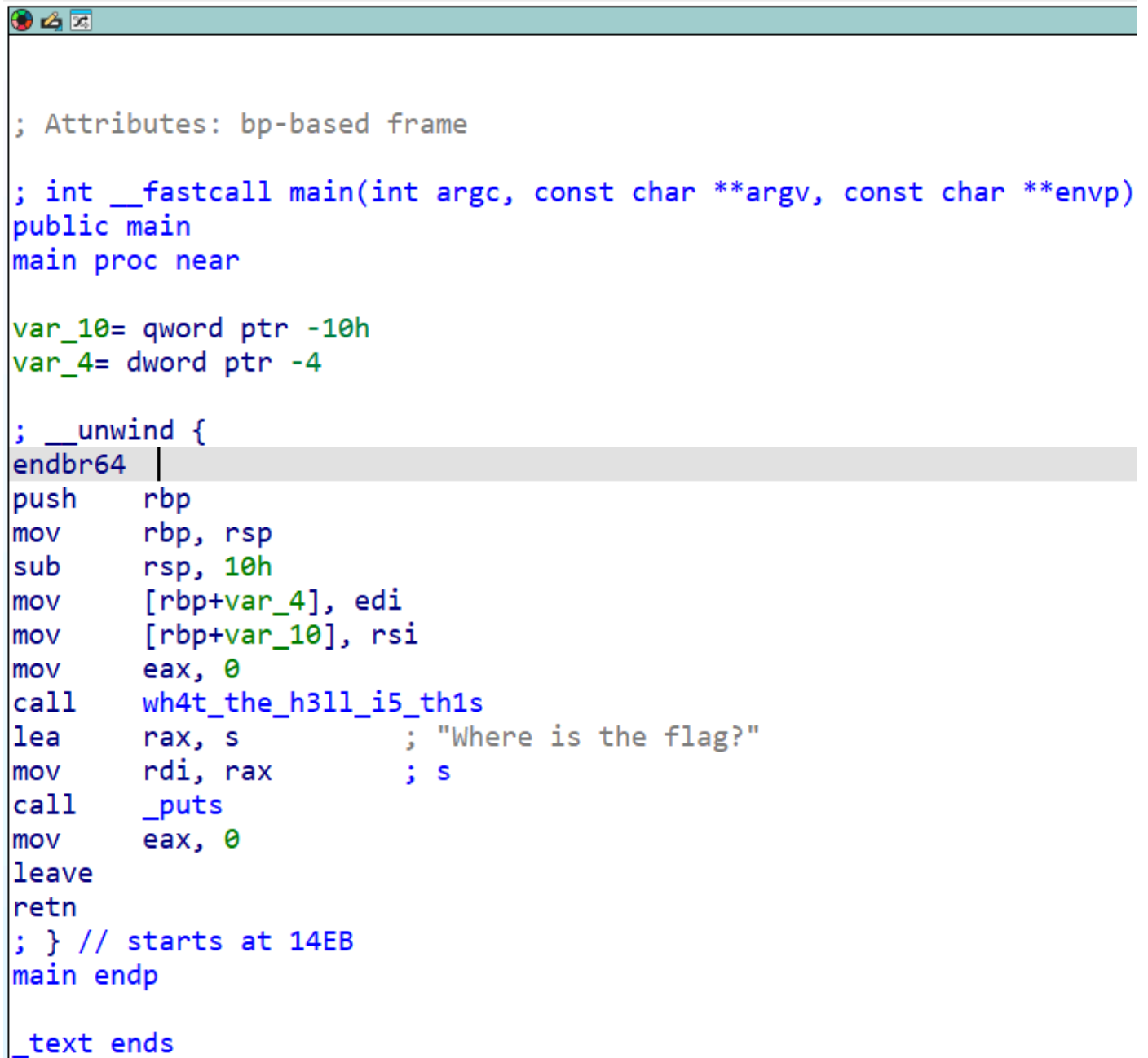
删除后重新用as生成就不会报错了, 而且最后生成的可执行文件可以正常运行:

```
→ Reverse gcc hello_clang_gcc.o -o hell.elfo
→ Reverse ./hell.elfo
Hello, World!
```

我的猜测是: addrSIG和.addrSIG\_sym是clang独有的, 因此as无法理解, 究其原因, 两者使用了同样的asm语法, 但两者使用的汇编器并不是同一个, clang中大概是集成了属于自己的汇编器工具。

## 2. Task 2

在IDA中对文件进行逆向，观察main函数发现调用了一个用户自定义函数：



```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10= qword ptr -10h
var_4= dword ptr -4

; __unwind {
endbr64 |
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
mov     eax, 0
call    wh4t_the_h3ll_i5_th1s
lea     rax, s          ; "Where is the flag?"
mov     rdi, rax        ; s
call    _puts
mov     eax, 0
leave
retn
; } // starts at 14EB
main endp

_text ends
```

转到该函数后可以发现又调用了一个函数：00000000

```
; Attributes: bp-based frame

public wh4t_the_h3ll_i5_th1s
wh4t_the_h3ll_i5_th1s proc near
; __unwind {
endbr64
push     rbp
mov      rbp, rsp
lea      rax, fl4g
mov      rdi, rax
call     00000000
nop
pop      rbp
retn
; } // starts at 14D1
wh4t_the_h3ll_i5_th1s endp
```

转到下一个函数后发现一行带有字符的汇编代码：

```
mov     byte ptr [rax], 41h ; 'A'
```

f	oo0o0o0
f	oo0o0oo
f	oo0oo00
f	oo0oo0o
f	oo0ooo0
f	oo0oooo
f	ooo0000
f	ooo000o
f	ooo00o0
f	ooo00oo
f	ooo0o00
f	ooo0o0o
f	ooo0ooo
f	oooo000
f	oooo00o
f	oooo0o0
f	oooo0oo
f	ooooo00
f	ooooo0o
f	oooooo0
f	ooooooo
f	wh4t_the_h3ll_i5_th1s
f	main

不断重复以上步骤，在每次调用的函数内找到一个字符，按顺序组合得到：  
AAA{hope\_u\_have\_fun~}，这就是需要找到的flag。

### 3. Task 3

---

下载的文件不能直接逆向，需要先通过clang输出.o文件，再进行逆向得到：

```

int __fastcall main(int argc, const char **argv, const char **envp)
{
    int v3; // eax
    int v5[32]; // [rsp+0h] [rbp-F0h] BYREF
    char s[72]; // [rsp+80h] [rbp-70h] BYREF
    const char **v7; // [rsp+C8h] [rbp-28h]
    int v8; // [rsp+D4h] [rbp-1Ch]
    int v9; // [rsp+D8h] [rbp-18h]
    int i; // [rsp+DCh] [rbp-14h]
    int j; // [rsp+E0h] [rbp-10h]
    int k; // [rsp+E4h] [rbp-Ch]

    v8 = 0;
    v9 = argc;
    v7 = argv;
    memset(s, 0, 0x40uLL);
    memset(v5, 0, sizeof(v5));
    _isoc99_scanf(&unk_5E0, s);
    for ( i = 0; i < strlen(s); ++i )
    {
        if ( s[i] < 48 || s[i] > 57 )
        {
LABEL_15:
            printf("try again\n");
            exit(0);
        }
    }
    for ( j = 0; j < strlen(s); j += 2 )
    {
        v3 = xcrc32(&s[j], 2, 0x414243u);
        v5[j / 2] = v3;
    }
    for ( k = 0; (unsigned __int64)k < 8; ++k )
    {
        if ( v5[k] != target[k] )
            goto LABEL_15;
    }
    printf("awesome\n");
    return 0;
}

```

通过代码逻辑判断，需要输入的字符串应当全是数字（ASCII码值在48和57之间）。输入的数字会经过一个xcrc32的处理：以两位为单位输入函数，经过转换后输出到一个数组，全部数字转换完成之后将得到的数字与target数组中的数字一一比对，完全一致的情况下才会输出awesome。

根据k的取值知道需要比对的次数是8，对应的输入数字长度应为16，现在需要找到xcrc32的转换逻辑。

打开函数得到:

```
__int64 __fastcall xcrc32(_BYTE *a1, int a2, unsigned int a3)
{
    while ( a2-- )
        a3 = crc32_table[(unsigned __int8)(*a1++ ^ HIBYTE(a3))] ^ (a3 << 8);
    return a3;
}
```

可以看出这是一个CRC32的操作,通过查表法进行转换,因此我们可以得到以下思路:通过每两位的穷举计算出对应的crc32值,并与target中的数字比对,需要的遍历次数只有800次,理论上是可行的。

在代码的数据部分得到crc32\_table和target的值,并以此写一个程序来爆破对应的数字串:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef unsigned char _BYTE;
#define HIBYTE(x) (((x) >> 24) & 0xFF)
unsigned int crc32_table[]={...};
unsigned int target[]={...};

__int64 __fastcall xcrc32(_BYTE *a1, int a2, unsigned int a3)
{
    while ( a2-- )
        a3 = crc32_table[(unsigned __int8)(*a1++ ^ HIBYTE(a3))] ^ (a3 << 8);
    return a3;
}

int main(){

    __int8 i = 0;
    __int8 j = 0;
    __int8 k = 0;
    unsigned int v5;

    for (k = 0; k < 8; k++)
    {
        for(i = 0; i < 10; i++){
            for(j = 0; j < 10; j++){
                _BYTE *v4 = (_BYTE *)malloc(2);
                v4[0] = i + '0';
                v4[1] = j + '0';
                v5 = xcrc32(&v4[0], 2, 0x414243u);
                if(v5 == target[k]){
                    printf("%dFound: %x %x\n", k, i, j);
                }
            }
        }
    }
}
```

```

    }
}
}

return 0;
}

```

代码中的数据部分已略去，运行后就能得到正确的输入：2012102420240704

```

0Found: 2 0
1Found: 1 2
2Found: 1 0
3Found: 2 4
4Found: 2 0
5Found: 2 4
6Found: 0 7
7Found: 0 4

```

在程序中验证通过：

```

55 ,0x4c11db70
56 ,0x48d0c6c7
57 ,0x4593e01e
58 ,0x4152fda9
59 ,0x5f15adac

```

问题 2 输出 调试控制台 终端 端口 注释

```

ine-Error-2wyk3l2w.gxh' '--pid=Microsoft-MIEngine-P
Exe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=m
0Found: 2 0
1Found: 1 2
2Found: 1 0
3Found: 2 4
4Found: 2 0
5Found: 2 4
6Found: 0 7
7Found: 0 4
PS D:\CS>

```

```

→ Reverse ls
challenge2.bc
challenge2.bc.old
challenge2.bc.old:Zone.Identifier
challenge2.bc:Zone.Identifier
→ Reverse ./challenge2.elf
2012102420240704
awesome
→ Reverse
→ Reverse
→ Reverse
→ Reverse

```

因此flag为AAA{2012102420240704}

## 4. Bonus

对程序进行逆向，在内部找到了三个main函数，但其中只有一个是真正的主函数。

```

f _io_stdin
f frame_dummy
f main
f main1
f main2
f handleError

```

其中一个假main函数使用了两次异或操作，并与数组中存的结果进行比对，用以下脚本进行恢复：

```

flag = [0x72, 0x71, 0x65, 0x76, 0x4C, 0x75, 0x54, 0x5A, 0x64, 0x43, 0x56, 0x4D,
0x60, 0x58, 0x54, 0x52, 0x47, 0x7D, 0x55, 0x48, 0x42, 0x79, 0x51, 0x56, 0x5E, 0x4F,
0x76, 0x4E, 0x43, 0x4F, 0x4A, 0x13, 0x6E]
data = []
for j in range(len(flag)):
    for i in range(0x20, 0x7f):
        if (i ^ j ^ 0x33) == flag[j]:
            original_value = i
            break
    data.append(chr(original_value))
print("flag=", ''.join(data))

```

运行后得到假flag: ACTF{Can\_you\_find\_the\_true\_flag?}

另外一个假main函数是对输入字符串进行base64加密，并与加密后的字符串比对。这里的base64用的顺序是自定义的，而且加密后还有一个根据序号进行移位的操作。根据以上得到以下脚本：

```

encoded_str = "OSLSPlrSYBDxVxDxV0Dqa07dWT7EKerjV0XqWUb7"
Base64list = '-*ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'
decoded_str = ''
for i, char in enumerate(encoded_str):
    index = Base64list.find(char)
    new_index = (index - i) % len(Base64list)
    new_char = Base64list[new_index]
    decoded_str += new_char




print(decoded_str)

```

这个代码输出的是恢复移位后的base64密文，使用cyberchef实现解密，得到：



Recipe



From Base64

Alphabet

-\*ABCDEFGHIJKLMNOP...

☒ Remove non-alphabet chars

☐ Strict mode

Input

OSLSP1rSYBDxVxDxV0Dqa07dWT7EKerjV0XqWUb7

REC 40

≡ 1

Output

ACTF{Th1s\_1s\_also\_a\_F0ke\_flag}

如上显示，这个flag依旧是一个假flag。

最后一个main是真main，但也是最复杂的一个。

```

void __fastcall __noreturn main1(__int64 a1, __int64 a2, __int64 a3)
{
    __int64 v3; // rdx
    unsigned int v4; // eax
    __int64 v5; // rsi
    __int64 v6; // rdx
    char v7; // [rsp+3h] [rbp-9Dh]
    int i; // [rsp+4h] [rbp-9Ch]
    unsigned int j; // [rsp+8h] [rbp-98h]
    unsigned int v10; // [rsp+Ch] [rbp-94h]
    char v11[48]; // [rsp+20h] [rbp-80h] BYREF
    char v12[72]; // [rsp+50h] [rbp-50h] BYREF
    unsigned __int64 v13; // [rsp+98h] [rbp-8h]

    v13 = __readfsqword(0x28u);
    puts("Welcome to ACTF2020!", a2, a3);
    puts("Input your flag here and I will check it for you: ", a2, v3);
    for ( i = 0; i <= 46; ++i )
    {
        v7 = getchar();
        if ( v7 == 10 || v7 == 13 )
            break;
        v11[i] = v7;
    }
    v11[i] = 0;
    v4 = j_strlen_ifunc(v11);
    v5 = v4;
    v10 = encrypt((__int64)v11, v4, (__int64)"_is_this_the_true_main()_?", (__int64)v12);
    for ( j = 0; v10 > j; ++j )
    {
        v6 = (int)j;
        if ( v12[j] != c1[j] )
        {
            puts("Try again! You can do it!", v5, (int)j);
            exit(0LL);
        }
    }
    puts("Aha, you get it!", v5, v6);
    exit(0LL);
}

```

encrypt函数内以12个字符为一组进行加密。加密过程中用了随机数生成器，但输入是确定的，可以得到值。然而里面用了很多的其他函数，如rho、theta，内部有许多移位操作。因此想彻底完成这道题的话，有相当大的工作量，所以没能全部完成。