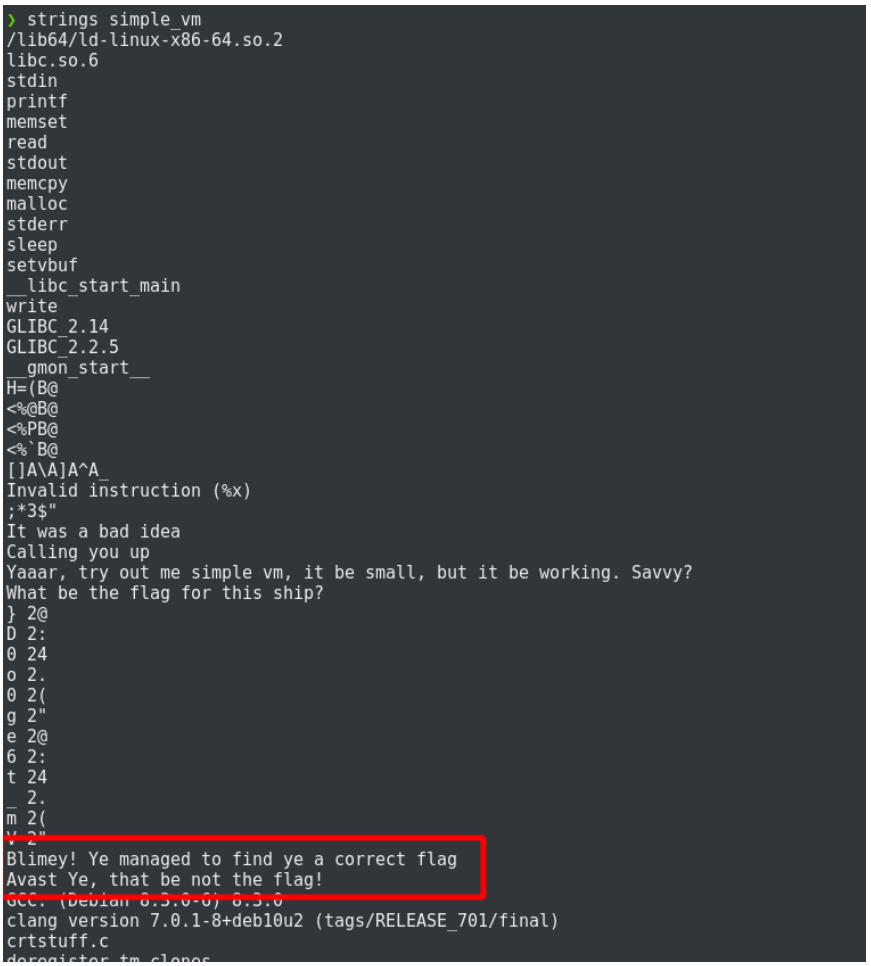
We are given a binary, that waits a bit, asks for a flag, then exits

Tools

- Ghidra
- Strings

Running strings on the binary, we can see two interesting ones



Looking at these two strings, we can see the error conditions for the file

Opening this up in ghidra and looking at the main function, we can see

```
1
2  undefined4 main(void)
3
4  {
5    void *pvVar1;
6
7    setvbuf(stdout,(char *)0x0,2,0);
8    setvbuf(stdin,(char *)0x0,2,0);
9    setvbuf(stderr,(char *)0x0,2,0);
10   pvVar1 = init_vm(&instructions);
11   while (*(int *)((long)pvVar1 + 0x508) != 0) {
12     step((long)pvVar1);
13   }
14   return 0;
15 }
16
```

That were is an init_vm, and a step function
Combining this with the description, we can see this is a vm challenge of a sort.

Looking into init_vm,

Decompile: init_vm - (simple_vm)

```
1
2  void * init_vm(void *param_1)
3
4  {
5    void *__s;
6
7    __s = malloc(0x50c);
8    memset((void *)((long)__s + 0x104),0,0x400);
9    memset(__s,0,0x100);
10   *(undefined4 *)((long)__s + 0x100) = 0;
11   *(undefined4 *)((long)__s + 0x508) = 1;
12   memcpy((void *)((long)__s + 0x104),param_1,0x400);
13   return __s;
14 }
15
```

We can immediately see the struct in there, renaming the variables a bit, we get

Decompile: init_vm - (simple_vm)

```
1
2  void * init_vm(void *param_1)
3
4  {
5    unknown_struct *s;
6
7    s = (unknown_struct *)malloc(0x50c);
8    memset(&s->param_field,0,0x400);
9    memset(s,0,0x100);
10   s->field256_0x100 = 0;
11   s->field1285_0x508 = 1;
12   memcpy(&s->param_field,param_1,0x400);
13   return s;
14 }
15
```

Not immediately helpful, other than the memcpy on the param_field
Looking into that param, we can see that it gets memcpy'ed into a field which I am calling param field.

Looking into param_1, on the first image, we can see that it was a reference to instructions

```
1
2  undefined4 main(void)
3
4  {
5    void *pvVar1;
6
7    setvbuf(stdout,(char *)0x0,2,0);
8    setvbuf(stdin,(char *)0x0,2,0);
9    setvbuf(stderr,(char *)0x0,2,0);
10   pvVar1 = init_vm(&instructions);
11   while (*(int *)((long)pvVar1 + 0x508) != 0) {
12     step((long)pvVar1);
13   }
14   return 0;
15 }
16
```

Looking into instructions, this looks like potentially the instructions that the vm executes.

```
                    instructions                                    XREF[3]:     Ent
                                                                                 maj
004040a0 01              ??         01h
004040a1 59 61 61        ds         "Yaaar, try out me simple vm, it be small, but...
         61 72 2c
         20 74 72 ...
004040e6 00              ??         00h
004040e7 45              ??         45h    E
004040e8 00              ??         00h
004040e9 01              ??         01h
004040ea ff              ??         FFh
004040eb 00              ??         00h
004040ec 00              ??         00h
004040ed 00              ??         00h
004040ee 08              ??         08h
004040ef 20              ??         20h
004040f0 30              ??         30h    0
004040f1 12              ??         12h
004040f2 00              ??         00h
004040f3 01              ??         01h
004040f4 00              ??         00h
004040f5 02              ??         02h
004040f6 ff              ??         FFh
004040f7 01              ??         01h
004040f8 2e              ??         2Eh    .
004040f9 20              ??         20h
004040fa 00              ??         00h
004040fb 00              ??         00h
004040fc 03              ??         03h
004040fd 00              ??         00h
004040fe 01              ??         01h
004040ff ff              ??         FFh
00404100 10              ??         10h
00404101 31              ??         31h    1
00404102 eb              ??         EBh
00404103 01              ??         01h
00404104 0a              ??         0Ah
00404105 00              ??         00h
00404106 00              ??         00h
00404107 02              ??         02h
00404108 00              ??         00h
00404109 01              ??         01h
0040410a ff              ??         FFh
0040410b 02              ??         02h
0040410c 01              ??         01h
0040410d 57 68 61        ds         "What be the flag for this ship? "
```

There seems to be two of the strings that we saw earlier, and apart from that, just random gibberish.

Also something also interesting, if we look

```
1
2  undefined4 main(void)
3
4  {
5    void *pvVar1;
6
7    setvbuf(stdout,(char *)0x0,2,0);
8    setvbuf(stdin,(char *)0x0,2,0);
9    setvbuf(stderr,(char *)0x0,2,0);
10   pvVar1 = init_vm(&instructions);
11   while (*(int *)((long)pvVar1 + 0x508) != 0) {
12     step((long)pvVar1);
13   }
14   return 0;
15 }
16
```

At the main function, we can see if the if comparison with the while, which is on an offset of 0x508, which is extremely similar to the offset we can for the init
Cleaning main up and renaming some functions

Decompile: main - (simple_vm)

```
1
2  undefined4 main(void)
3
4  {
5    vm_struct *machine_struct;
6
7    setvbuf(stdout,(char *)0x0,2,0);
8    setvbuf(stdin,(char *)0x0,2,0);
9    setvbuf(stderr,(char *)0x0,2,0);
10   machine_struct = (vm_struct *)init_vm(&instructions);
11   while (machine_struct->on_field != 0) {
12     step((long)machine_struct);
13   }
14   return 0;
15 }
16
```

Now we can see the functionality of the program which seems to be continuously stepping over the struct while a certain field is set.
If we now look into the step function, we can see all of the if statements.

Decompile: step - (simple_vm)

```
1
2  void step(vm_struct *_vm_struct)
3
4  {
5    byte bVar1;
6    char cVar2;
7    char cVar3;
8    int iVar4;
9    int iVar5;
10
11   bVar1 = (&_vm_struct->param_field)[*(int *)&_vm_struct->field_0x504];
12   if (bVar1 == 0xcc) {
13     _vm_struct->on_field = 0;
14   }
15   else if (bVar1 == 0xff) {
16     iVar4 = pop((long)_vm_struct);
17     cVar2 = (char)iVar4;
18     iVar4 = pop((long)_vm_struct);
19     cVar3 = (char)iVar4;
20     if (cVar2 == '\0') {
21       read(0,&_vm_struct->field_0x0 + (int)_vm_struct->field256_0x100,(long)cVar3);
22       _vm_struct->field256_0x100 = (int)cVar3 + _vm_struct->field256_0x100;
23     }
24     else if (cVar2 == '\x01') {
25       write(1,(void *)((long)_vm_struct + ((long)(int)_vm_struct->field256_0x100 - (long)(
26                        ),(long)cVar3);
27       _vm_struct->field256_0x100 = _vm_struct->field256_0x100 - (int)cVar3;
28     }
29     else if (cVar2 == '\x02') {
30       sleep((int)cVar3);
31     }
32   }
33   else if (bVar1 == 1) {
34     do {
35       iVar4 = *(int *)&_vm_struct->field_0x504 + 1;
36       *(int *)&_vm_struct->field_0x504 = iVar4;
37       cVar2 = (&_vm_struct->param_field)[iVar4];
38       push((long)_vm_struct,cVar2);
39     } while (cVar2 != '\0');
40   }
41   else if (bVar1 == 0) {
42     iVar4 = *(int *)&_vm_struct->field_0x504 + 1;
43     *(int *)&_vm_struct->field_0x504 = iVar4;
44     push((long)_vm_struct,(&_vm_struct->param_field)[iVar4]);
45   }
46   else if (bVar1 == 2) {
```

Scrolling down, we can see more if statements.

Now, since we know that this is a vm, we can deduce that bVar1 would be the instruction type, and that field_0x504 would be the ip.

The increment at the bottom confirms this

```
    _vm_struct->running = 0;
  }
  _vm_struct->ip = _vm_struct->ip + 1;
  return;
}
```

If we now look into the pop function, we can see that the decompiler mistakes the arguments as a long

Decompile: pop - (simple_vm)

```
1
2  int pop(long param_1)
3
4  {
5    int iVar1;
6
7    iVar1 = *(int *)(param_1 + 0x100) + -1;
8    *(int *)(param_1 + 0x100) = iVar1;
9    return (int)*(char *)(param_1 + iVar1);
10 }
11
```

When actually it should be the vm_struct as seen from

```
lse if (instruction == 0x20) {
 iVar3 = pop((long)_vm_struct);
 iVar4 = pop((long)_vm_struct);
 push((long)_vm_struct,(char)iVar4
```

Changing this in the function reveals quite a lot of information to us.

Decompile: pop - (simple_vm)

```
1
2  int pop(vm_struct *param_1)
3
4  {
5    int iVar1;
6
7    iVar1 = param_1->field256_0x100 + -1;
8    param_1->field256_0x100 = iVar1;
9    return (int)(char)(&param_1->field_0x0)[iVar1];
10 }
11
```

We can see that field256_0x100 is decremented, then set back into the param.
We can then see the first field being indexed as an int/char.
Retyping it as a char, improves the decompilation

Decompile: pop - (simple_vm)

```
1
2  char pop(vm_struct *param_1)
3
4  {
5    int iVar1;
6
7    iVar1 = param_1->field256_0x100 + -1;
8    param_1->field256_0x100 = iVar1;
9    return (&param_1->field_0x0)[iVar1];
10 }
11
```

```
else if (instruction == 0x20) {
  cVar1 = pop(_vm_struct);
  cVar2 = pop(_vm_struct);
  push((long)_vm_struct,cVar2);
```

Currently based on the name, and what the functionality of the function, it looks like this is operating on a stack, however to confirm this, let's look at the push function

Decompile: push - (simple_vm)

```
1
2  void push(long param_1,undefined param_2)
3
4  {
5    int iVar1;
6
7    iVar1 = *(int *)(param_1 + 0x100);
8    *(int *)(param_1 + 0x100) = iVar1 + 1;
9    *(undefined *)(param_1 + iVar1) = param_2;
10   return;
11 }
12
```

Looks similar to how pop was before we improved it, adjusting some of the names and types, we get

Decompile: push - (simple_vm)

```
1
2  void push(vm_struct *_vm_struct,char push_var)
3
4  {
5    int current_sp;
6
7    current_sp = _vm_struct->field256_0x100;
8    _vm_struct->field256_0x100 = current_sp + 1;
9    (&_vm_struct->field_0x0)[current_sp] = push_var;
10   return;
11 }
12
```

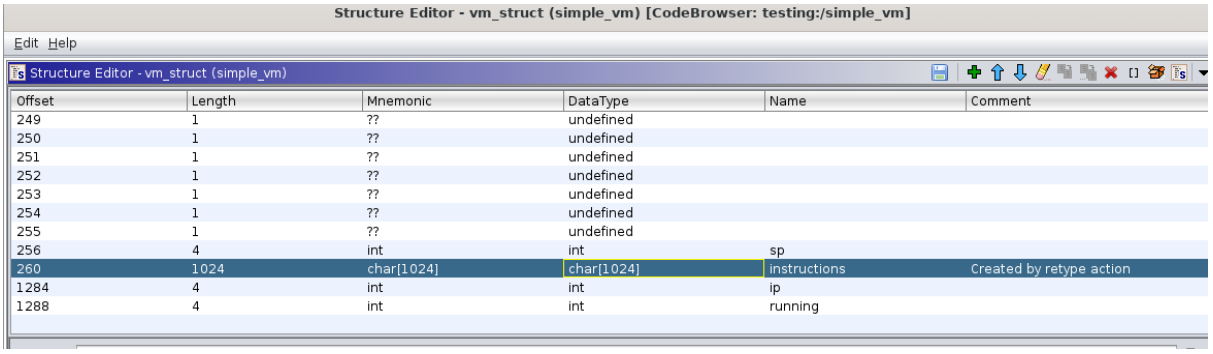
It seems like the function is pushing onto a char array or byte array, so we can assert that field256_0x100 should be the stack pointer.

Looking at the vm_struct now, we can see that the sp seems to separate two different areas of the struct. If we go back now to the init_vm function, it seems to make much more sense what is happening.

Decompile: init_vm - (simple_vm)

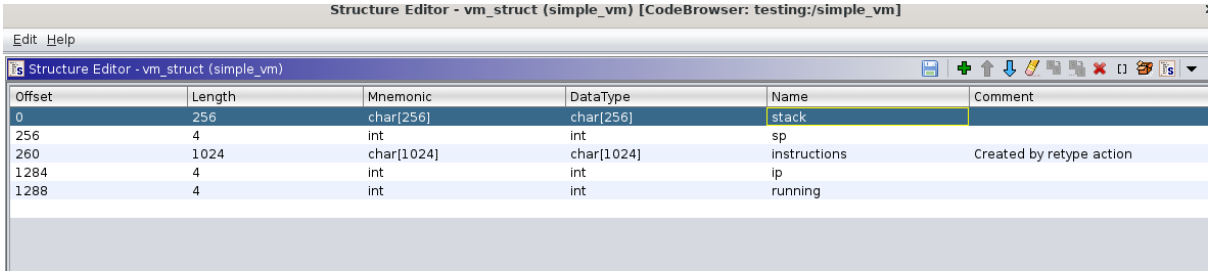
```
1
2  vm_struct * init_vm(void *param_1)
3
4  {
5    vm_struct *s;
6
7    s = (vm_struct *)malloc(0x50c);
8    memset(&s->instructions,0,0x400);
9    memset(s,0,0x100);
10   s->sp = 0;
11   s->running = 1;
12   memcpy(&s->instructions,param_1,0x400);
13   return s;
14 }
```

Seeing that instructions seems to be memcpy'ed of 0x400, we can try to set it to a char array of 0x400 and see what happens



Offset	Length	Mnemonic	DataType	Name	Comment
249	1	??	undefined		
250	1	??	undefined		
251	1	??	undefined		
252	1	??	undefined		
253	1	??	undefined		
254	1	??	undefined		
255	1	??	undefined		
256	4	int	int	sp	
260	1024	char[1024]	char[1024]	instructions	Created by retype action
1284	4	int	int	ip	
1288	4	int	int	running	

We can see it works cleanly.
Looking at the second memset starting from 0 and going to 0x100, and assuming that its the stack, we get:



Offset	Length	Mnemonic	DataType	Name	Comment
0	256	char[256]	char[256]	stack	
256	4	int	int	sp	
260	1024	char[1024]	char[1024]	instructions	Created by retype action
1284	4	int	int	ip	
1288	4	int	int	running	

Which in my opinion, is extremely clean and satisfying

Now, since we have done the majority of 'difficult' reversing, we need to now actually figure out.

I won't go into detail specifically on how to solve the actual flag, but I will give a high level overview on one way to do it.

If you reverse the opcodes, eventually you will see a push, cmp pattern for the flag.
These are in 6 byte intervals each, so once you figure that out, it's pretty trivial to find the flag, also seeing that the middle is slightly inconsistant

The sleep at the start is to discourage any potential automated ways of solving this, as that is not the intended solution.