Return-oriented shellcoding. As part of a larger attack, you need to use ROP to create some code to call the Linux `mprotect` system call and disable W⊕X protection on the rest of your shellcode. In particular you've figured out that the call you want to make would look like:

```
mprotect(stack_pointer - 10000, 20000, PROT_WRITE|PROT_EXEC);
```

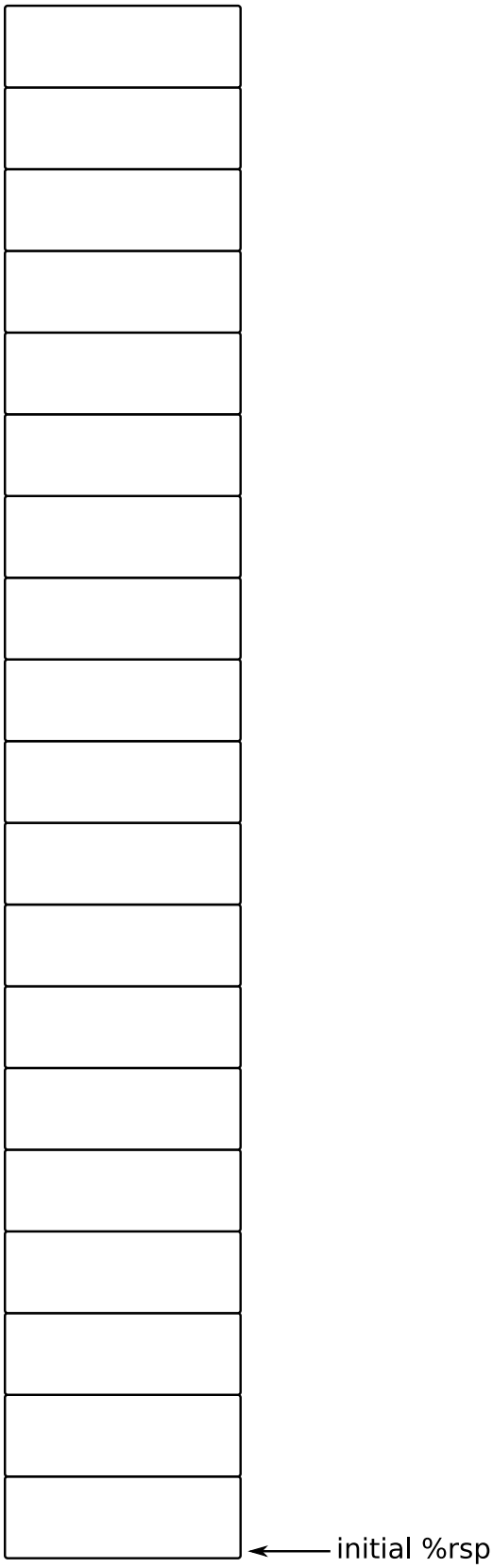
if you could write it in C. (`stack_pointer` can be the value of `%rsp` at any point in the shellcode, since the margin of 10000 bytes on either side is enough to take care of any minor variation.) Looking up in the appropriate header files, you've also discovered that `mprotect` is system call number 125, and that the numeric value of the protection flags is $2|4 = 6$. According to the Linux system call calling conventions, you need to put the system call number in `%rax`, put the three arguments in the registers `%rdi`, `%rsi`, and `%rdx` respectively, and then execute the instruction `syscall`.

You've also found a number of useful-looking gadgets, which are shown on the right side of the next page. Your job is to fill in the return-oriented program on the picture of the stack shown on the left side of the next page, starting from the "top" of the stack at the bottom of the page. Each space on the stack represents a 64-bit value. You can fill it in with a fixed number by writing the number in the box, or you can make it a pointer to a gadget by writing the letter of the gadget in the box. (When drawing these diagrams in the past we've drawn an arrow from the box to the gadget, but letters should be easier to read.) You can use each gadget as many or as few times as you would like: our solution uses all of them, some several times.

Hint: plan carefully the order in which you fill in the registers, since some operations can only be done using certain registers for intermediate values.

It's possible to achieve your goal using only the gadgets shown, and without using all the stack spaces we've drawn for you. But if you can't figure out how, you can attempt a solution in which you invent new gadgets (write them below the ones we put there with their own letters).

We've written the x86 instructions in AT&T syntax, so the result operands are on the end. For instance `mov x, y` is like `y = x`, and `add x, y` is like `y += x`.



A. `syscall; ret`

B. `mov %rsp, %rdi; ret`

C. `add %rdi, %rax; ret`

D. `mov %rsi, %rax; ret`

E. `pop %rsi; ret`

F. `mov %rax, %rdi; ret`

G. `mov %rax, %rdx; ret`