Paxson Spring 2017 — CS 161 Computer Security — Homework 1

Due: Thursday, January 26, at 11:59pm

Instructions. This homework is due **Thursday, January 26, at 11:59pm**. No late homeworks will be accepted. This assignment must be done on your own.

Create an EECS instructional class account if you have not already. To do so, visit https://inst.eecs.berkeley.edu/webacct/, click "Login using your Berkeley CalNet ID," then find the cs161 row and click "Get a new account." Be sure to take note of the account login and password, and log in to your instructional account.

Make sure you have a Gradescope account and are joined in this course. The homework *must* be submitted electronically via Gradescope (not by any other method). Your answer for each question, when submitted on Gradescope, should either be a separate file per question, or a single file with each question's answer on a separate page.

Problem 1 ***Policy*** **(10 points)**

The aim of this exercise is to ensure that you read the course policies, as well as to make sure that you are registered in the class and have a working EECS instructional class account.

Open the course website https://www.icir.org/vern/cs161-sp17/.
Append `?lastname=<name>&userid=cs161-xy` to the URL in the address bar, where `<name>` is your last name (as in campus records) and `cs161-xy` is your class ID (with `xy` replaced with the two final characters of your class account). (If you have spaces or apostrophes in your last name, go ahead and type them in: they should not cause any problems.) Thus, the URL you will open is:

https://www.icir.org/vern/cs161-sp17/?lastname=<name>&userid=cs161-xy

Please read and check that you understand the course policies on that page. If you have any questions, please ask for clarification on Piazza.

To receive credit for having read the policies, submit the following statement as your answer for Q1: "`I understand the course policies.`" (no quotes necessary)

Solution: The correct answer is:

```
I understand the course policies.
```

:-)

Problem 2 ***Policy*** **(10 points)**

You're working on a course project. Your code isn't working, and you can't figure out

why not. Is it OK to show another student (who is not your project partner) your draft code and ask them if they have any idea why your code is broken or any suggestions for how to debug it?

Read the course policies carefully. Based on them, determine the answer to this question, and submit it for Q2 on Gradescope. A one-word answer is fine: we do not need a detailed explanation (though you may provide an explanation if you choose).

Solution: No, it is never acceptable to show another student your code, even if it is just a draft.

Problem 3 ***Hidden Backdoor*** **(30 points)**

We have hidden a secret password (a "backdoor") on the web page you visited in Question 1. If you entered the URL correctly as above (substituting your last name and userid), you'll be able to access the password—if you know the trick.

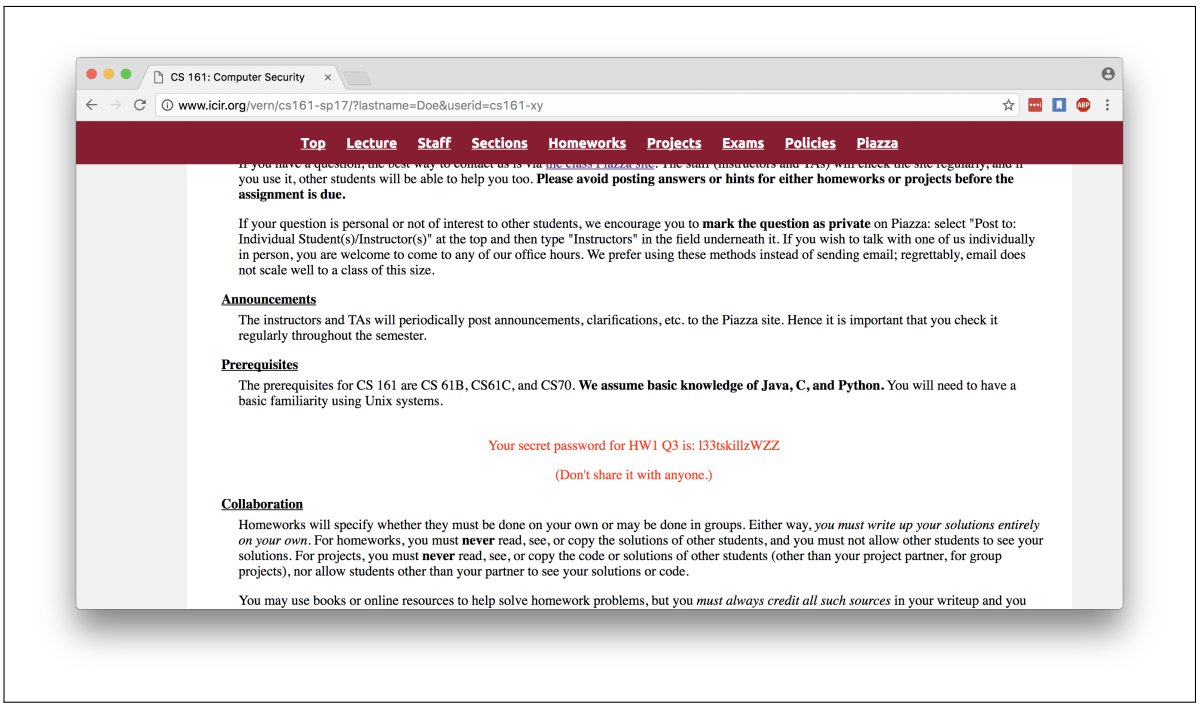
What's the trick? You're going to have to figure that out on your own. But we'll help you out: we will share a hint on Piazza. Make sure you have set up your Piazza account (using the same name on Piazza as the University has for you in its records), which you can do by following the link provided on the course web page. Then, go look for the post on Piazza where we disclose the hint.

Once you have figured out the password, submit just the password as your answer to Q3 on Gradescope. In accordance with the class policies on doing work individually, do not share the password with anyone.

Solution: Enter the Konami code on the course webpage by pressing the following keys:

```
UP UP DOWN DOWN LEFT RIGHT LEFT RIGHT B A
```

Your password will appear just before the section of the web page on collaboration.



Problem 4 *Feedback* (0 points)

Optionally, feel free to include feedback. What's the single thing we could do to make the class better? Or, what did you find most difficult or confusing from lectures or the rest of class, and what would you like to see explained better? If you have feedback, submit your comments as your answer to Q4.

Solution: Feedback is always welcome!

Problem 5 *Memory Layout* (50 points)

Consider the following C code:

```
1  int dog(int i, int j, int k) {
2      int t = i + k;
3      char *newbuf = malloc(4);
4      t = t + 4;
5      t = t - j;
6      return t;
7  }
8
9  void cat(char *buffer) {
10     int i[2];
11     i[1] = 5;
12     i[0] = dog(1, 2, i[1]);
13 }
14
15 int main() {
16     char buf[8];
17     cat(buf);
18     return 0;
19 }
```

The code is compiled and run on a 32-bit x86 architecture (i.e., IA-32). Assume the

program is run until line 6, meaning everything before line 6 is executed (i.e., a breakpoint was set at line 6). We want you to sketch what the layout of the program's stack looks like at this point. In particular, print the template provided on the next page and fill it in. Fill in each empty box with the value in memory at that location. Put down specific values in memory, like 1 or 0x00000000, instead of symbolic names, like buf. Also, on the bottom, fill in the values of %ebp and %esp when we hit line 6. Submit the filled template as your answer to Q5 on Gradescope.

Assumptions you should make:

- memory is initially all zeros
- execution starts at the very first instruction during the usual invocation of main(), and %esp and %ebp start at 0xa0000064 at that point
- the call to malloc returns the value 0x12341234
- the address of the code corresponding to line 4 is 0x01111180
- the address of the code corresponding to line 13 is 0x01111134
- the address of the code corresponding to line 18 is 0x01111100
- a char is 1 byte, an int is 4 bytes
- no function uses general-purpose registers that need to be saved (other than %ebp)

See the next page for the template.

Solution: A good way to figure out the memory layout of this process is to walk through, step by step, what is happening.

Preliminary Recall that a processor accesses memory at the granularity of *memory words*. In 32-bit x86, also called IA-32, the size of a memory word is 32 bits, or 4 bytes. Each "cell" in the stack diagram will therefore be one word, or 4 bytes.

Also, remember that the stack starts at a high address, and grows downwards towards lower addresses.

Executing Main() As stated in the assumptions, %esp and %ebp start with the value 0xa0000064. We know that the stack pointer (%esp) always points to the bottom of the stack, e.g., the last allocated word on the stack. From this, we know that the first non-allocated word is at 0xa0000060, which is the top word/cell in our stack diagram.

The assumptions also state that the execution starts at the very first instruction during the usual invocation of *main*. This means we start with the first instruction of the function prologue, shown below:

```
push %ebp      // save the previous frame pointer
mov %esp %ebp  // start a new frame by moving EBP down to ESP
sub X %esp     // X = total size of local variables (word-aligned)
```

Stepping through this, we see that the value of the %ebp register (which is 0xa0000064 as stated in the assumption) is the first thing pushed onto the stack. Then %ebp is moved down to point to this word in the second instruction. Finally, we must make space for the local variable(s) of *main*. In this case, we have a 8-character buffer, which is 8-bytes (or 2 words). So X in the third instruction is $8, and we move the stack pointer down two words. Since *main* does not set the value of these characters, the memory value will be initially 0, given our assumptions.

This results in the stack so far looking as shown below:

Address	**Memory Value**
0xa0000060	0xa0000064 (sfp)
0xa000005c	0 (buf[4-7])
0xa0000058	0 (buf[0-3])

At this point, %ebp = 0xa0000060 and %esp = 0xa0000058.

Next, we call the *cat* function from *main*.

Executing Cat When calling a function, we'll need to put the function arguments onto the stack, and save the return instruction pointer (*rip*). Then we proceed with the called function's prologue, as usual.

cat takes the array *buf* as an argument. Recall that an array variable's value is the address of the first element in the array, and the first element is stored at the lowest address. Thus, *buf* = 0xa0000058, and this value is pushed onto the stack as the called function's argument.

Next we save *rip* during the *call* assembly instruction, which stores the address of the next instruction to execute after returning from the called function. In this case, line 18 of main() is the next line executed. As the assumptions state, this line is at 0x01111100, the address next pushed onto the stack. Now our stack looks like:

Address	**Memory Value**
0xa0000060	0xa0000064
0xa000005c	0
0xa0000058	0
0xa0000054	*0xa0000058 (buf)*
0xa0000050	*0x01111100 (rip)*

Here, %ebp = 0xa0000060 and %esp = 0xa0000050.

Again, we execute the function prologue for *cat*, which will store the current %ebp

value of 0xa0000060 onto the stack, and make enough space to hold the i int array. This array will require 2 words, both initialized to 0 (per our assumptions). However, the second element at the higher address will be set to the value 5.

Address	**Memory Value**
0xa0000060	0xa0000064
0xa000005c	0
0xa0000058	0
0xa0000054	0xa0000058
0xa0000050	0x01111100
0xa000004c	*0xa0000060 (sfp)*
0xa0000048	*5 (i[1])*
0xa0000044	*0 (i[0])*

Now, %ebp has been updated to point to the current stack frame's base at 0xa000004c (where the old %ebp value is stored) and %esp = 0xa0000044.

Almost there! Now, we call *dog* in *cat*.

Executing dog Executing *dog* is similar to executing *cat*. We again need to store the following data, in order:

1. **Arguments:** *dog* requires 3 int arguments, which requires 3 words of memory. The first argument is always at the lowest address, and the last argument is at the highest. Therefore, we push the arguments for *dog* onto the stack in reverse order, pushing 5, 2, then 1.

2. **rip:** *rip* stores the next line to execute once we return from *dog* which is line 13. As stated, this line is at address 0x01111134.

3. **sfp:** After saving *rip*, we again perform the function prologue which starts with pushing %ebp onto the stack. Thus its current value of 0xa000004c is pushed on, and %ebp will be set to that word's address (0xa0000030).

4. **Local Variables:** Again, the function prologue ends by allocating enough space for local variables. For *dog*, this is an int and a character pointer, which together take up 2 words. In reality, a compiler may order these local variables in different orders (and we'll accept either orders as correct answers). However, we'll assume that the int t is at the higher address, and the pointer $newbuf$ is at the lower. We see that through *dog*, t is updated to ultimately equal $i + k + 4 - j = 8$. Also $newbuf$ is updated to the return value of *malloc*, which is 0x12341234 from the assumptions.

At this point, we hit the breakpoint, and are done. The stack now looks as below:

Address	Memory Value
0xa0000060	0xa0000064
0xa000005c	0
0xa0000058	0
0xa0000054	0xa0000058
0xa0000050	0x01111100
0xa000004c	0xa0000060
0xa0000048	5
0xa0000044	0
0xa0000040	*5 (arg 3)*
0xa000003c	*2 (arg 2)*
0xa0000038	*1 (arg 1)*
0xa0000034	*0x01111134 (rip)*
0xa0000030	*0xa000004c (sfp)*
0xa000002c	*8 (int t)*
0xa0000028	*0x12341234 (char* newbuf)*

As we worked through above, the **final** %ebp = 0xa0000030, and %esp = 0xa0000028 (the bottom of our stack).

DONE!

Final Thoughts You might be wondering why there doesn't appear to be anything on the stack from the call to *malloc*. Remember that *malloc* already returned by our breakpoint! This means all the data local to *malloc*, including its arguments, data stored by *malloc*'s function prologue, and whatever local values *malloc* had, all get deallocated upon its return. The only lasting data is the return value, which is stored in register %eax, not the stack, before being moved to the location of $newbuf$.

Note: If you happened to have written the memory values in for *malloc*, but have the correct stack pointer (indicating everything below the stack pointer is no longer a valid part of the stack), you will receive full credit.

The full solution, using the given template, is on the following page. This can be very tricky to first learn, so if you have questions, definitely ask on Piazza or visit office hours!

0xa0000060:	0xa0000064
0xa000005c:	0
0xa0000058:	0
0xa0000054:	0xa0000058
0xa0000050:	0x01111100
0xa000004c:	0xa0000060
0xa0000048:	5
0xa0000044:	0
0xa0000040:	5
0xa000003c:	2
0xa0000038:	1
0xa0000034:	0x01111134
0xa0000030:	0xa000004c
0xa000002c:	8***
0xa0000028:	0x12341234***
0xa0000024:	
0xa0000020:	
0xa000001c:	
0xa0000018:	
0xa0000014:	

.
.
.

% ebp = 0xa0000030

% esp = 0xa0000028

*** Might be switched. Under given constraints order of local variables is ambiguous.