CPSC 313: Computer Hardware and Operating Systems

Unit 1: The y86 (as a sequential processor)

2024 Winter Term 1

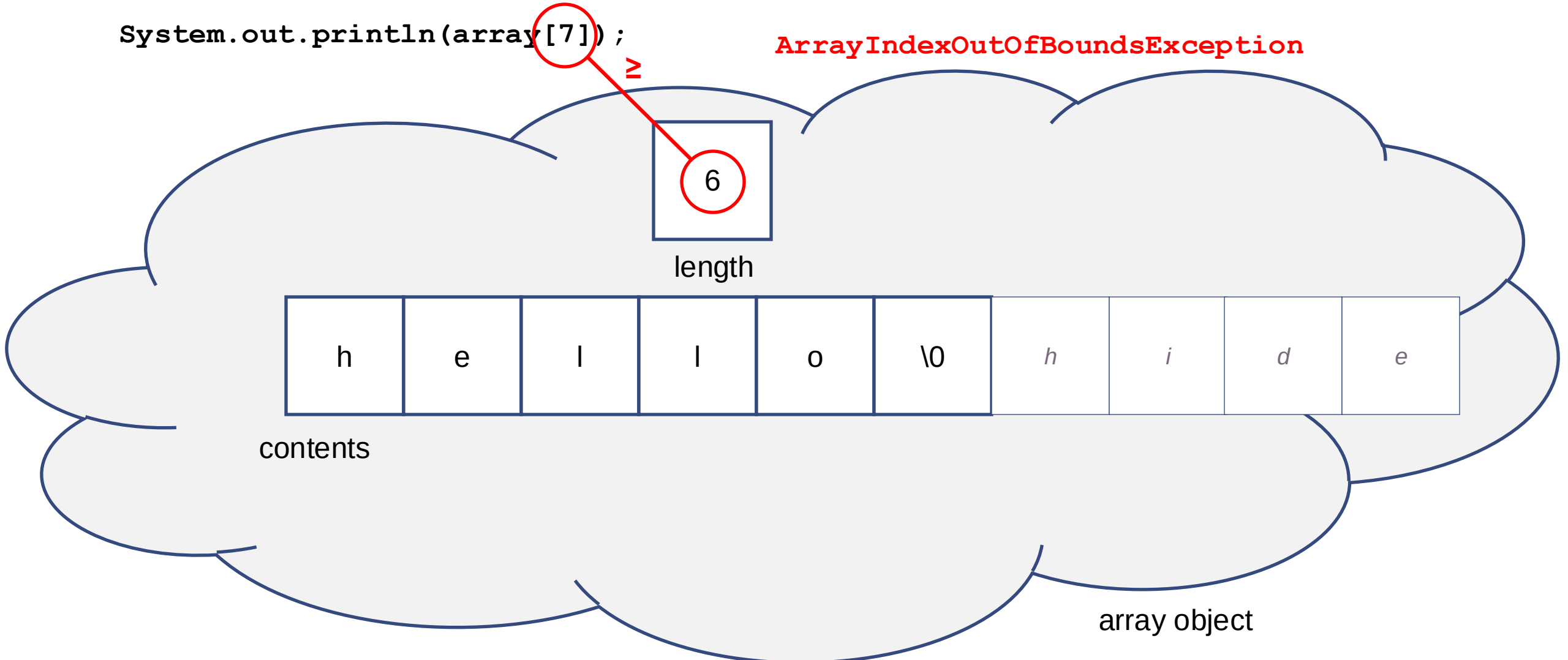
Admin

- Lab 1 due soon; Quiz 0 due Wednesday
- Lab 2 coming out soon
- Quiz 1 coming soon **reserve your slot on us.prarietest.com by today!**
  - You should also reserve your viewing and retakes!
    You can always cancel them later if you want.
- Second week of tutorial coming next week

Learning y86 via Buffer Overflows

- Today is going to be one in-class exercise designed to
  - Help you explore the y86 architecture
  - Be fun
  - Demonstrate how buffer overflows work
  - Help you identify how to prevent buffer overflow attacks
- You really, really need to understand the pre-class work for today!!!
- We will be using:
  - The simulator
  - Code snippets available here
  - Which points to the CPSC313-202341 git repo – 1.4-buffer-overflow

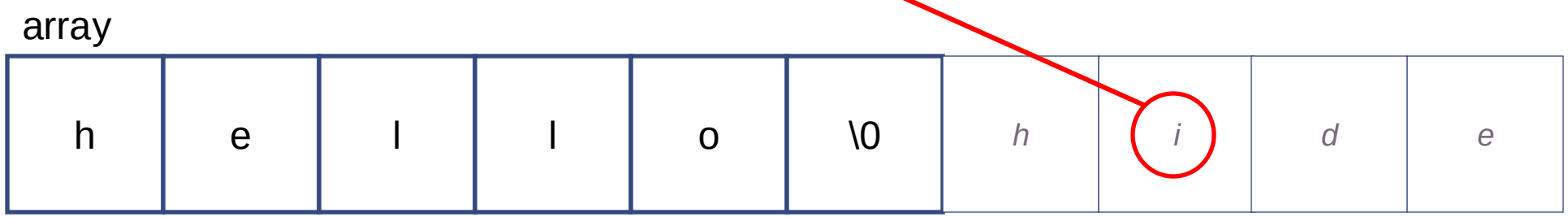
Out-of-Bounds Buffer Access: Java Array



Out-of-Bounds Buffer Access: C Array

```
printf("%c\n", array[7]);
```

Prints out `i`



Of course, we **can** protect against this in C, just as in Java,
nor does Java flawlessly stop all out-of-bounds accesses!

Key: Just because you *intend* a piece of data to be in some spot at some size,
doesn't mean everyone will do what you intend.

Buffer Overflows

- When we exploit the fact that you can (sometimes) write outside of a buffer, thus corrupting memory around it.
- Here are several examples of their consequences
  - The Internet Worm [1988]: a graduate student experiment gone wrong exploited a buffer overflow to infect 10% of the internet within 2 days. Major institutions and military systems were offline for days.
  - Heartbleed [2014, xkcd.com/1354]: a buffer overflow in OpenSSL that was exploited and resulted in more than $500,000,000 in damage.
  - WhatsApp VOIP [2019]: smartphones could be infected with malware by just calling the target phone (even if the call was not answered).

Step 1: Warm up: Writing an infinite loop

- Copy `01-exploit1.ys` into the simulator.

1. Add instructions for an infinite loop in the `exploit` function.
2. Add a `nop` into the loop so you can clearly see it running.
3. Save this file (using the Save File button).

Step 2: Warm up: Reading y86 Code

- Copy `01-exploit2.ys` into the simulator. It has a driver program and some data to exercise the function named `mystery`

1. Document what `mystery` does in comments
   - Read the code
   - Run it in the simulator
   - Determine the meanings of the parameters and of each register used
2. Give `mystery` a more descriptive name.
3. Save the file.

Step 3. Our First Exploit!

We'll make a benign function call a malicious one: `evil`. When `evil` returns, the program enters an infinite loop!

1. Run `01-exploit3.ys` in the simulator.
   - To what address does `evil` return?
   - Where is that return address stored?
2. Add your whole `exploit` function from Step 1 to the end of this program.
3. Add code to `evil` that overwrites the return address with the address of `exploit`. (This should take only 2 instructions!)
4. Run the code to check that calling `evil` returns… to `exploit`?!
5. Save your code.

Congratulations!!!!!

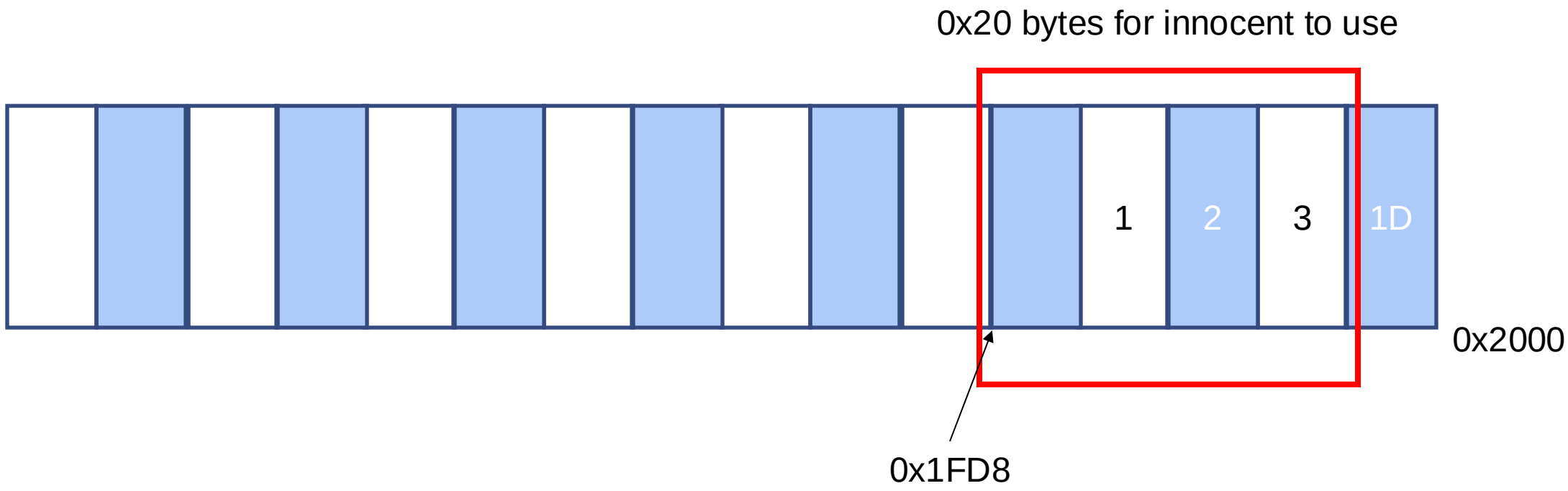
- You have written your first exploit.
- What precautions do high level languages use to prevent this exploit?

- We could avoid this by not invoking evil functions!
- But.. next, we have a malicious party exploit a poorly written but benign function: `mystery` from Step 2!

Step 4 Exploit, the “right” way.

- Copy the file `01-exploit4.ys` into the simulator

1. Draw a stack diagram at the moment a call to `innocent` is allocating its local variables.
2. In the diagram, identify where to allocate a buffer of 0x20 bytes that `innocent` can use.


0x20 bytes for innocent to use
1
2
3
1D
0x2000
0x1FD8

Step 4: Stack frame setup/teardown

Write the code to setup and teardown `innocent`'s stack frame with its 0x20 byte local buffer:

1. Find the PART 1 comment.
    1. Add code to setup a stack frame.
    2. Place the address of the buffer into `%rsi`.
2. Find the PART 2 comment. Add code to teardown the stack frame.
3. Run your code and ensure it does what you expect.

Step 5: Using the function from Step 2

- Add `exploit` to the end of this program as well.
- Add your renamed `mystery` function from Step 2 to the end of the program.
- Find PART 3 in the comments in the code
- Call the renamed `mystery` function.
- Step through your program:
  - What happens the first time through with `gooddata`?
  - What happens the second time through with `baddata`?
  - Why does this happen?

Exploit!

- In Step 3, you invoked `exploit` by overwriting the return address on the stack with its address (so `ret` returns to the wrong place).
- `baddata` caused the function to return to the wrong place, but only accidentally. Now, *do it on purpose*.
- Force the program to return to `exploit`, just by passing in evil **data**.
- This time, pass a pointer to data you create (`evildata`!!) that is designed to overwrite the return address with the address you prefer.

Step 6:Making things break in the “right” way.

1. Find PART 4 and uncomment the next two instructions.
2. Draw a stack diagram of the call to the renamed `mystery` function with:
   a) The local buffer
   b) The values on the stack if you know them
   c) Where `%rsi` and `%rdi` are pointing
3. Circle the spot where you want to place the address of `exploit`.
4. Now, fill in exactly as many values as needed under `evildata` so that when `innocent`, it returns to `exploit`!

Prevention

Question: What are the ways to prevent buffer overflow?

Wrapping Up

- Even though the y86 is simple, it is powerful enough to do “bad” things.
- **Use the skills and information you learn in this class only for good!**
- Next time: Implementing a y86 – its pre-class exercise is one of the few long collections of videos. If you did it a while ago, it is worth reviewing it before next class!