ECE458/ECE750T27: Computer Security
Programming Security

Dr. Kami Vaniea,
Electrical and Computer Engineering
kami.vaniea@uwaterloo.ca

First, the news…

- First 5 minutes we talk about something interesting and recent
- You will not be tested on the news part of lecture
- You may use news as an example on tests
- Why do this?
    1. Some students show up late for various good reasons
    2. Reward students who show up on time
    3. Important to see real world examples

Marks and Spencer data breach and ransomeware.

OR

How cybersecurity caused there to be no food.

BBC Home News Sport Earth Reel Worklife

NEWS

Home | Israel-Gaza war | War in Ukraine | Climate | Video | World | US & Canada | UK | Business | Tech

Tech

M&S hackers sent abuse and ransom demand directly to CEO

BLOOMBERG VIA GETTY IMAGES

Joe Tidy
Cyber correspondent, BBC World Service

6 June 2025

The Marks & Spencer hackers sent an abuse-filled email directly to the retailer's boss gloating about what they had done and demanding payment, BBC News has learnt.

Monday/Tuesday is Canada Day so there is no lecture Monday.

Wednesday there is lecture.

HISTORY … WHY ARE PROGRAMMING LANGUAGES SO INSECURE?

Toy dog I had as a kid

I “programmed” the design on its sleeping bag using punch cards

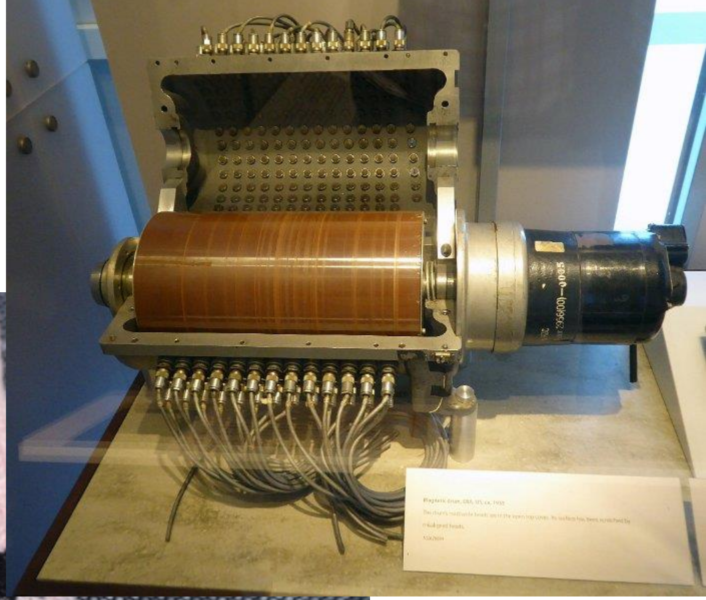
Knitting is a grid. Punch card says which color to use in each square

Understanding this programming language is easy. Whole system easy to understand and reason about. Programmer trusted to think about and avoid security/reliability problems without assistance.

My point:

New technology is built on old technology

Really old computer technology assumed the programmer had full understanding of the system



Modern programming assumes abstraction

OSI network model, for example, abstracts upper and lower layers

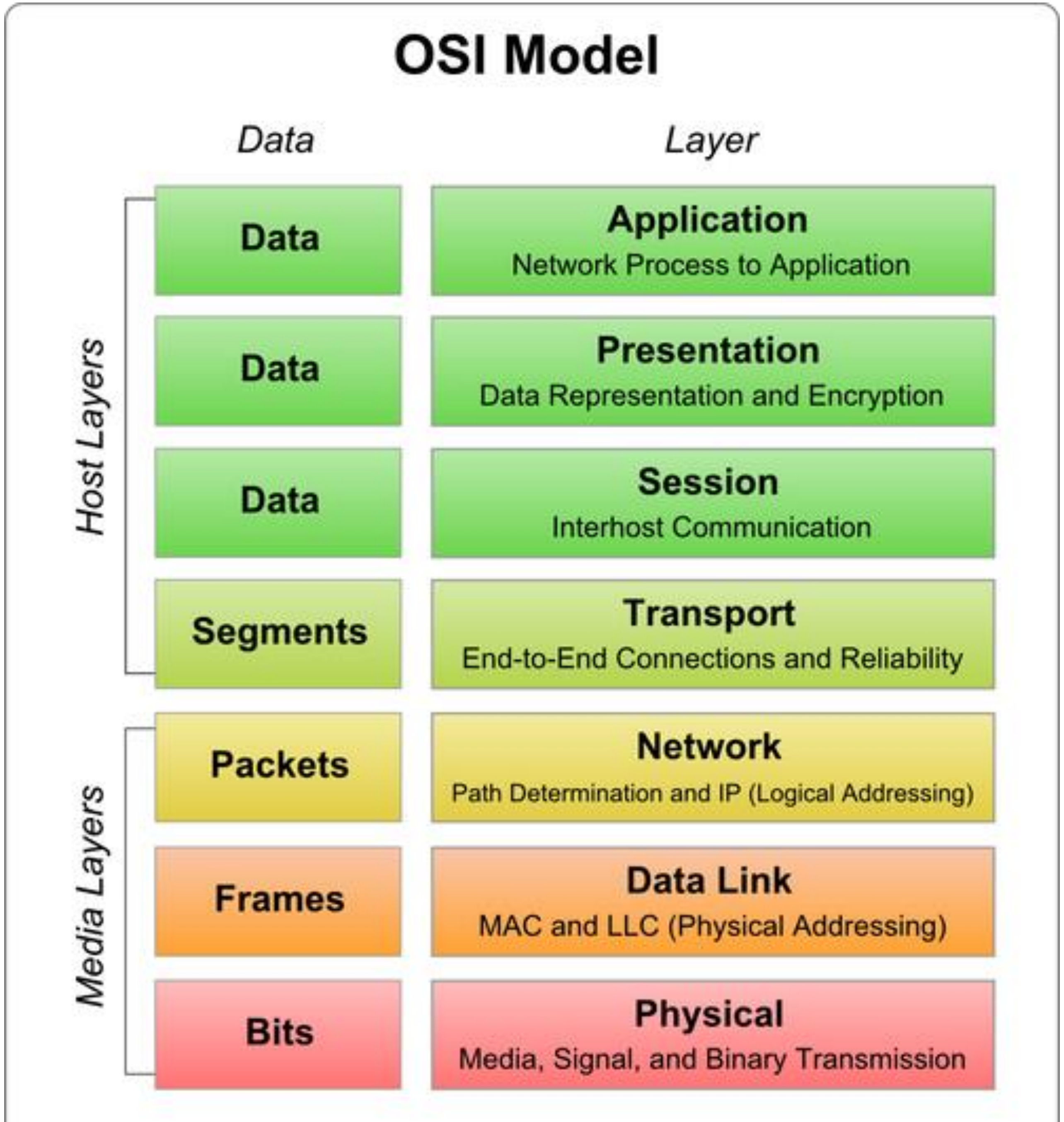


Image from: http://www.tech-faq.com/osi-model.html

SECURITY VS. PRIVACY AND TRUST

Theoretically how the system works

Actually how the system works

Security professional: Remove, block, defend, or otherwise prevent unintended harmful uses of a system.

Security expects specifications

- There is no such thing as a fully secure system that is protected from everything.
- Systems can be protected against specified threats, allow specified actions, and accept specified risks.

- **Confidential**: only authorized entities can read data or infer information
- **Integral**: only authorized entities can alter data.
- **Available**: authorized entities can access the data
- **Accountable**: all actions are recorded and traceable to who/what did it
- **Authenticated**: all entities have had their identities or credentials verified

How "secure" are the images on your smartphone?

1. Threat model – what threats are you worried about?
2. Access goal – who or what do you want to access your photos?
3. Risk acceptance – what risks are you ok with taking?
4. Trust model – who or what are you going to trust?
   (Even if it/they don't deserve it.)

- Based on the above: how secure are the images on your smartphone?

ERRORS VS FLAWS

Errors vs Flaws

Errors

- Unintentional
- Mistakes
- Typos
- Possible to find through testing
- If you were to compare a system diagram or specification to the code, there should be a discrepancy
- Can only be detected after programming

Flaws

- Code intentionally written that way, security side-effects likely unintentional
- Code follows intended design
- Hard to impossible to find from testing, if test and code are both based on the same system design plan
- Can be detected at the project planning phase

Error: Goto fail

- C-function called "SSLVerifySignedServerKeyExchange"
- Code checks the certificate validity of websites

```
OSStatus err;

 if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
     goto fail;
 if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
     goto fail;
 if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
     goto fail;
 if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
     goto fail;
     goto fail;
 if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
     goto fail;

 fail:
      return err;
```

https://slate.com/technology/2014/02/apple-security-bug-a-critical-flaw-was-extraordinarily-simple.html

Error: Goto fail

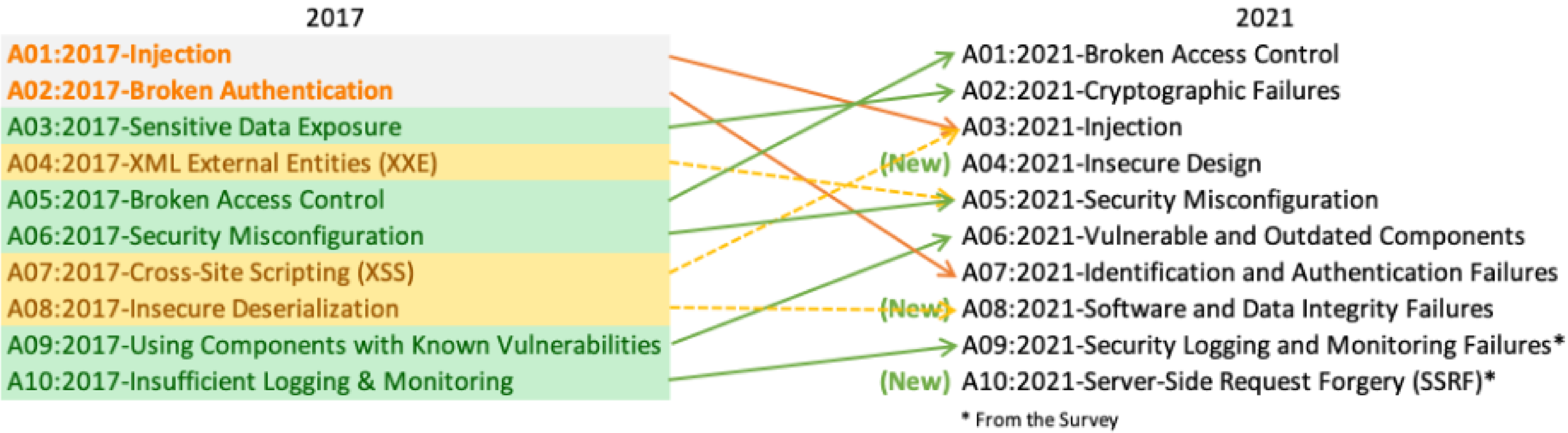
- C-function called "SSLVerifySignedServerKeyExchange"
- Code checks the certificate validity of websites
- The problem is the double "goto fail" which cause the last check to always be skipped
- Result: certificate validity was never checked… for 18 months on most Apple products

```
OSStatus err;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

fail:
    return err;
```

https://slate.com/technology/2014/02/apple-security-bug-a-critical-flaw-was-extraordinarily-simple.html

Sadly, the big problems change slowly

OWASP Top 10 Vulnerabilities



Sadly, the big problems change slowly

OWASP Top 10 Vulnerabilities

OWASP Top 10 - 2013	OWASP Top 10 - 2017
A1 – Injection	A1:2017-Injection
A2 – Broken Authentication and Session Management	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

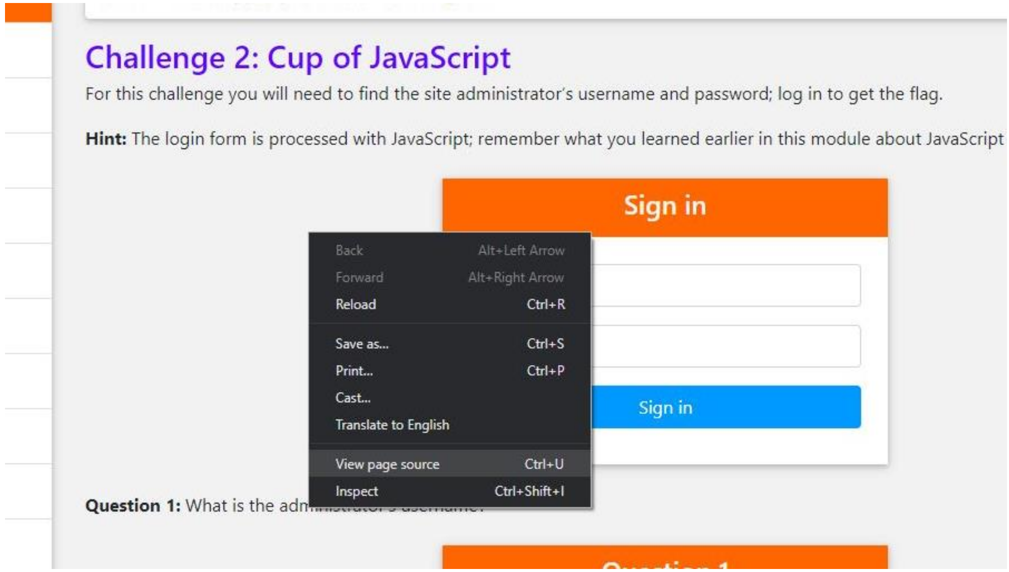
Top 10 Software Security Design Flaws

1. Earn or give but never assume trust
2. Use an authentication mechanism that cannot be bypassed or tampered with
3. Authorize *after* you authenticate
4. Strictly separate data and control instructions and never process control instructions received from untrusted sources
5. Define an approach that ensures all data are explicitly validated
6. Use cryptography correctly
7. Identify sensitive data and how they should be handled
8. Always consider the user
9. Understand how integrating external components changes your attack surface
10. Be flexible when considering future changes to objects and actors

Avoiding the top 10 software security design flaws by Iván Arce et al.

Flaw: Use an authentication mechanism that cannot be bypassed or tampered with.

- Example is from a Capture the Flag event
- Viewing page source shows that the password is hard coded in the page’s Javascript
- Giving the client the correct password so client-side authentication can be done is a design flaw



https://ctfacademy.github.io/web/challenge2/answer.htm

Flaw: Use an authentication mechanism that cannot be bypassed or tampered with.

- E e
- V p p
- C s d

```
<!-- Sign in form -->
<script> function buttonFunction() {


var myPassword = document.getElementById("password").value;

var username = document.getElementById("username").value;

var myalert = "637466617b436c69656e742d736964655f56616c69646174696f6e7d";


    var admin = "admin";

    var pass = "SuperSecretPassword";

 var myString = hex2a(myalert);

if (username == admin && myPassword == pass) {



               $("#loginFeedback").html("Great job! The flag is below.");
```

https://ctfacademy.github.io/web/challenge2/answer.htm

BUFFER OVERFLOW

Buffer Overflow

- More data is written into a buffer than the buffer has allocated memory
- As a result, the memory allocated next to the buffer gets overwritten
- For example, initiate a character array (A) and an unsigned short integer (B). Then copy a string into A that has a length > A. Result B is overwritten.

```
char           A[8] = "";
unsigned short B = 1979;
```

	A								B	
Value	null								1979	
Hex	0	0	0	0	0	0	0	0	07	BB

```
strcpy(A, "excessive");
```

Value	e	x	c	e	s	s	i	v	25856	
Hex	65	78	63	65	73	73	69	76	65	00

Buffer Overread

- More data is read from a buffer than the buffer has allocated memory
- As a result, the memory allocated next to the buffer gets read as well

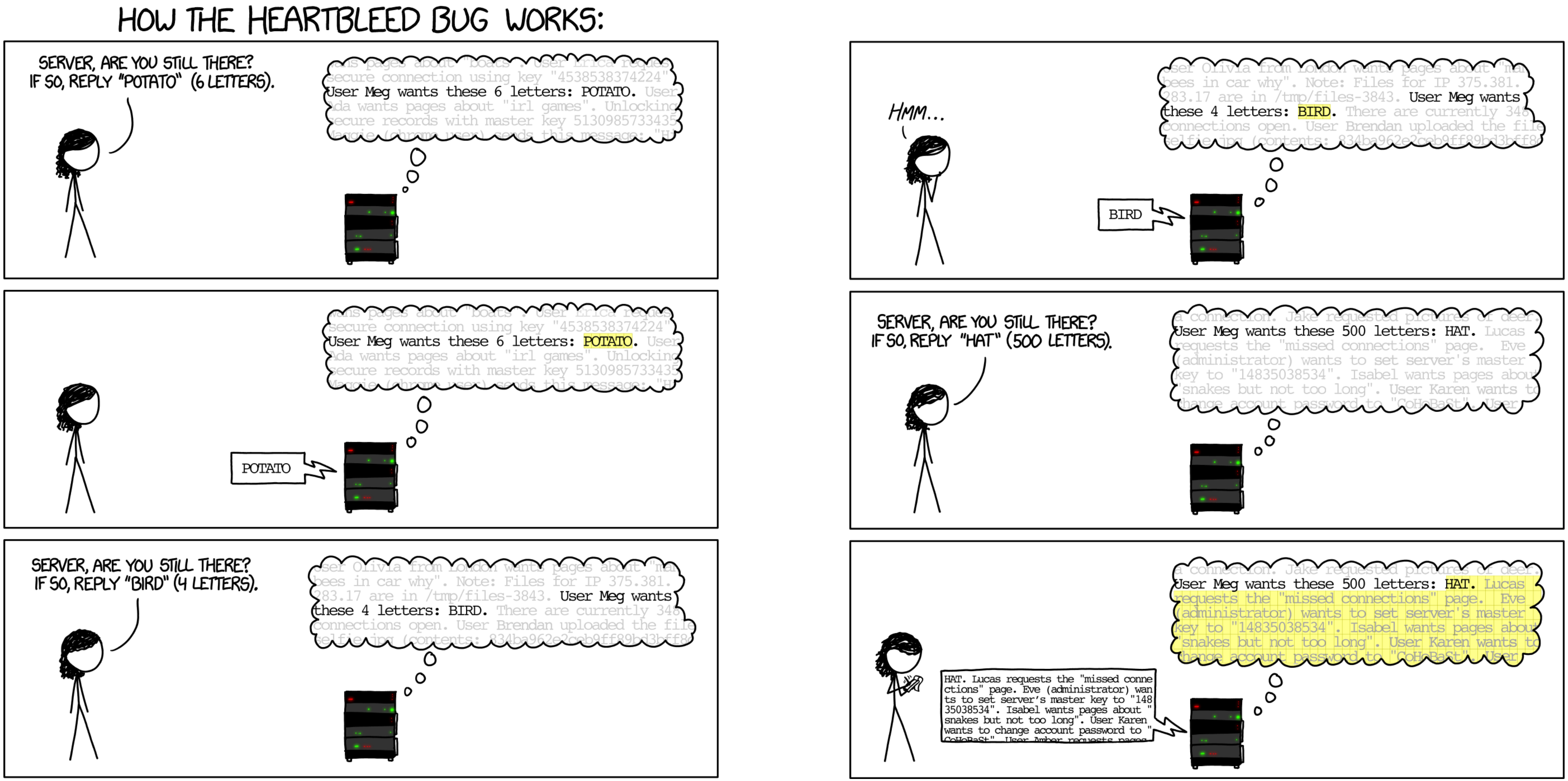
```
char           A[8]  = "";
unsigned short B = 1979;
```

	A								B	
Value	null								1979	
Hex	0	0	0	0	0	0	0	0	07	BB

```
char overread = *((char *)A + 8);
```

Value	e	x	c	e	s	s	i	v	25856	
Hex	65	78	63	65	73	73	69	76	65	00

Buffer Overread: Heartbleed (XKCD Comic)

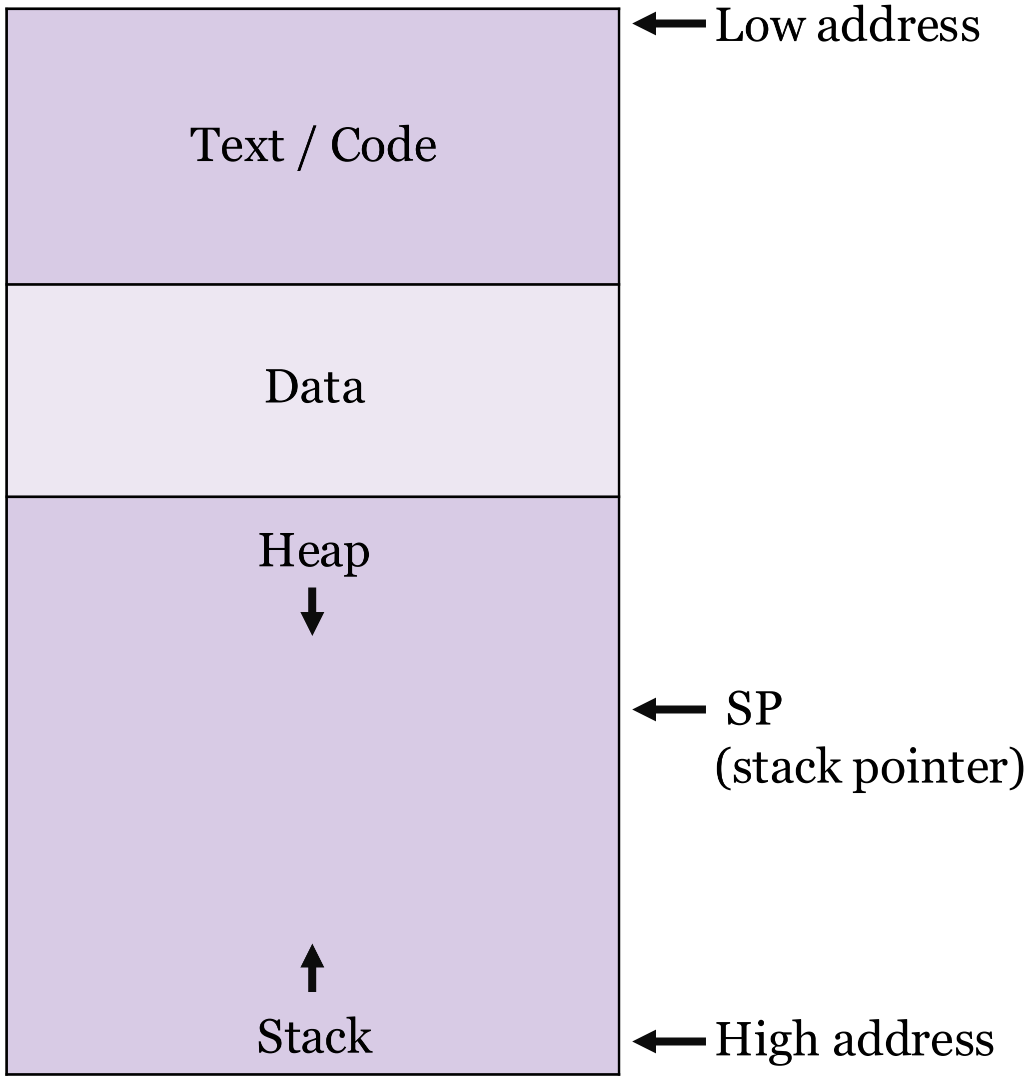


CALL STACK

Warning: I’m about to present the most common way memory is handled. Every OS, and compiler does it a bit differently.

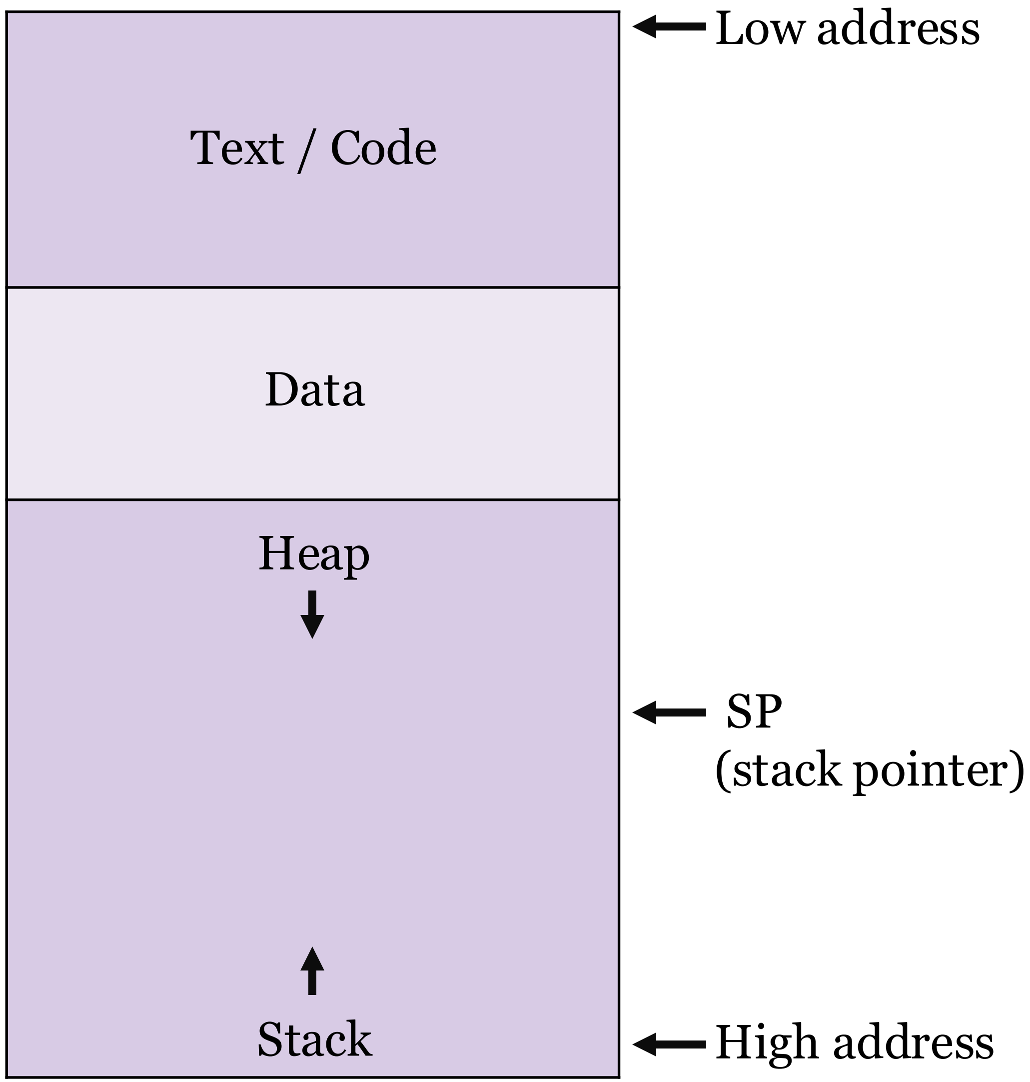
Memory Basics

- Memory for a program is roughly divided into:
  - Text/Code
  - Data – static variables
  - Heap – dynamic data (malloc)
  - Stack – processor “scratch paper” – dynamic local variables, parameters for functions, return address for function call
- Stack grows towards lower addresses
  - Stack Pointer (SP) tracks the “top” of the stack
- Heap grows towards bigger addresses



Think-pair-share

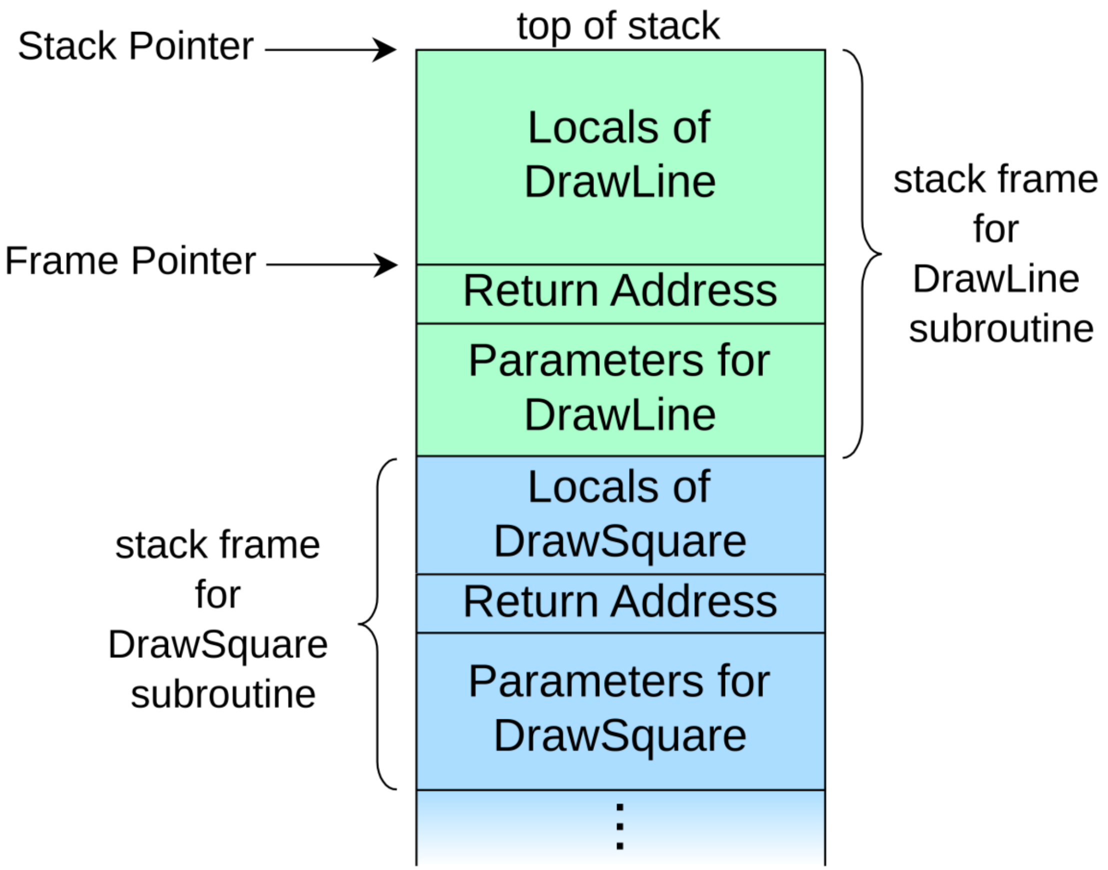
- What would happen if the Heap and the Stack ran into each other? (Assuming the OS didn’t notice/care.)
- How might an attacker use such a collision?



Call Stack

- When a new function is called, the current function:
  - Adds parameters to the top of the stack
  - Adds its own return address
  - Jumps to the new function

```
void DrawLine(int a, int b){
        char buffer[10];
}
void DrawSquare(int a, int b){
        DrawLine(1,2);
}
void main(){
        DrawSquare(1,2);
}
```

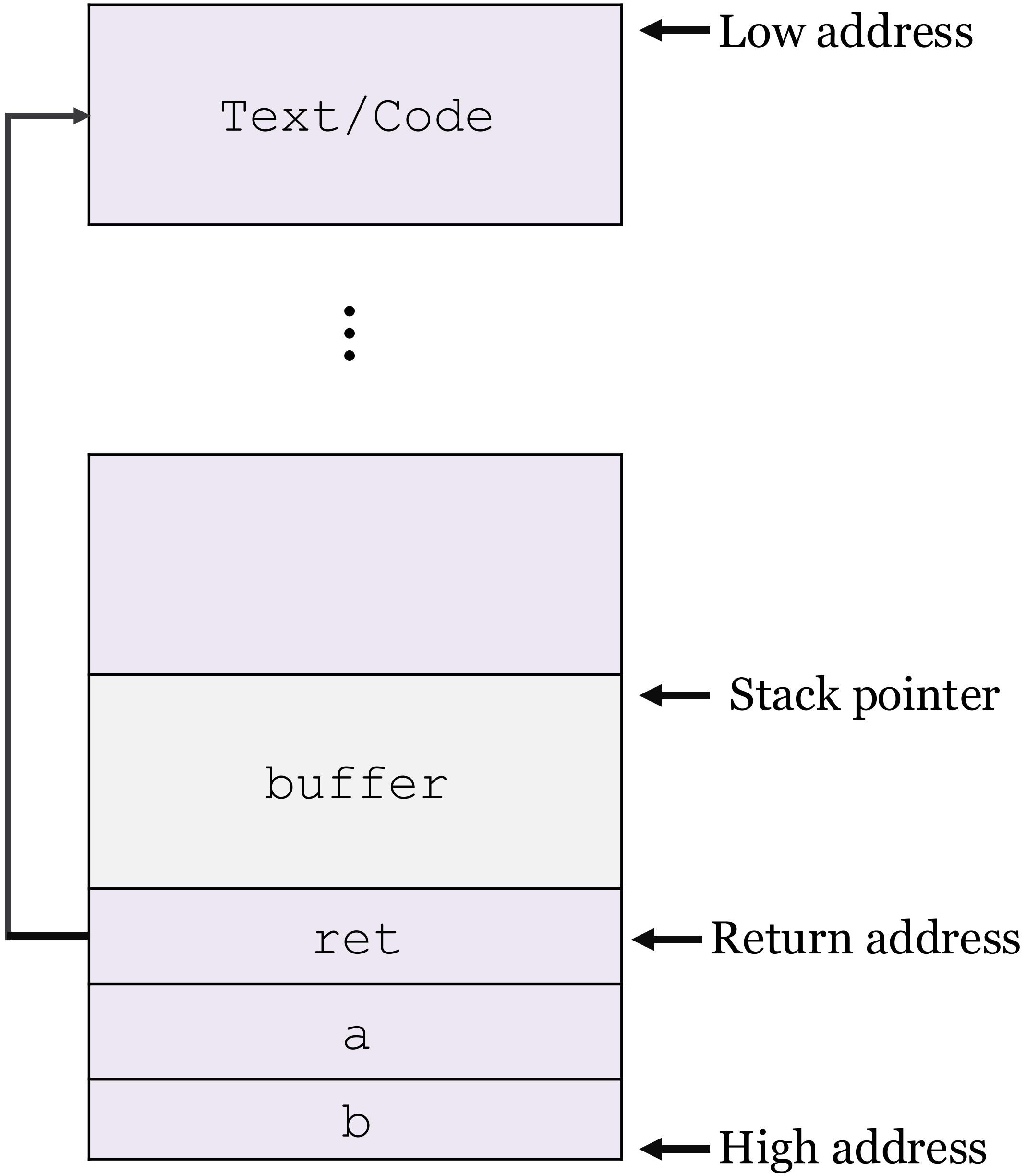


By R. S. Shaw - Wikipedia

Function call

- When a function is called, the calling function adds the function parameters to the stack and a return address

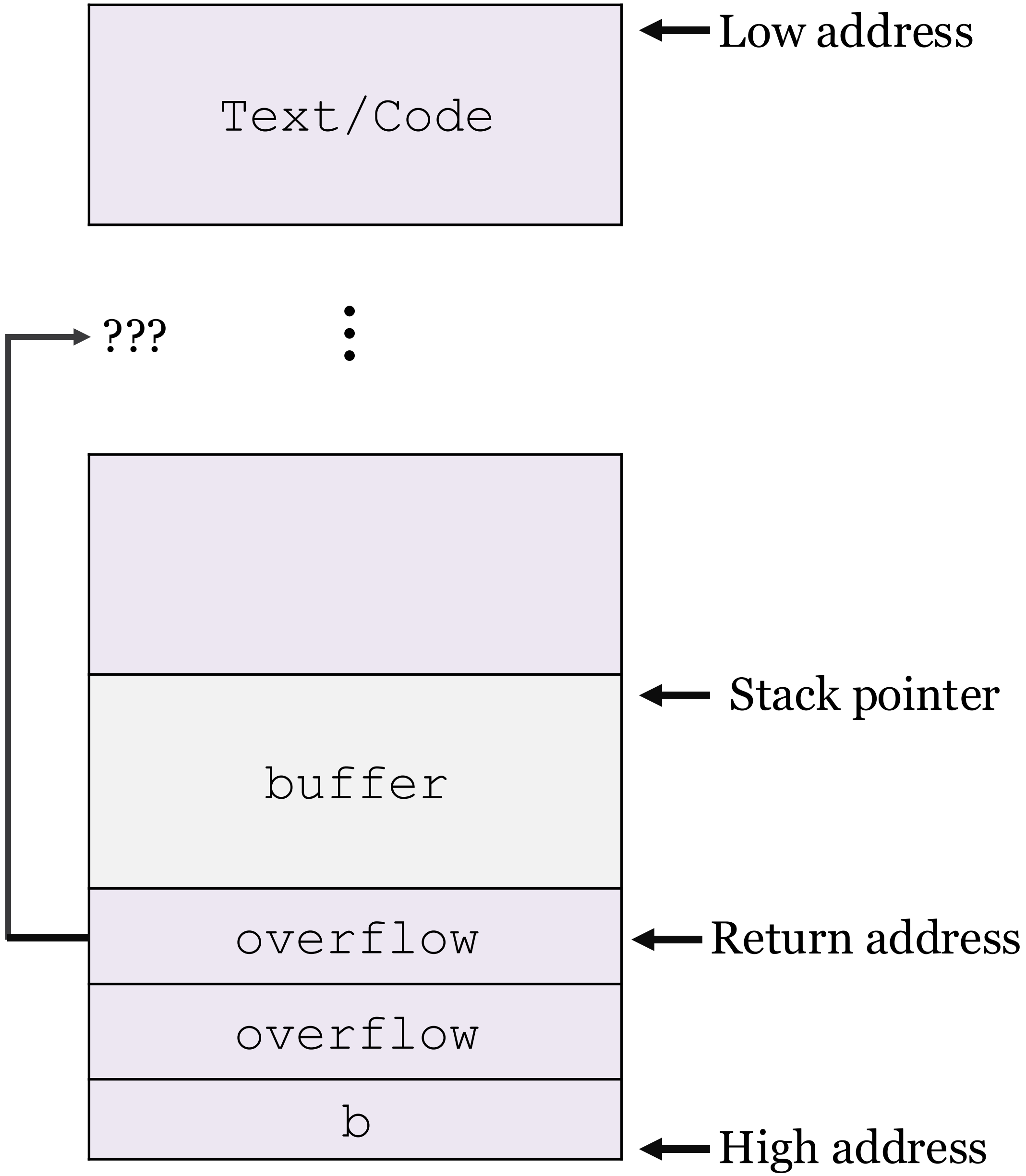
```
void func(int a, int b){
        char buffer[10];
}
void main(){
        func(1,2);
}
```



Stack smashing

- Goal: overwrite the return address
- Result
  - Crash the program
  - Execute code at new address

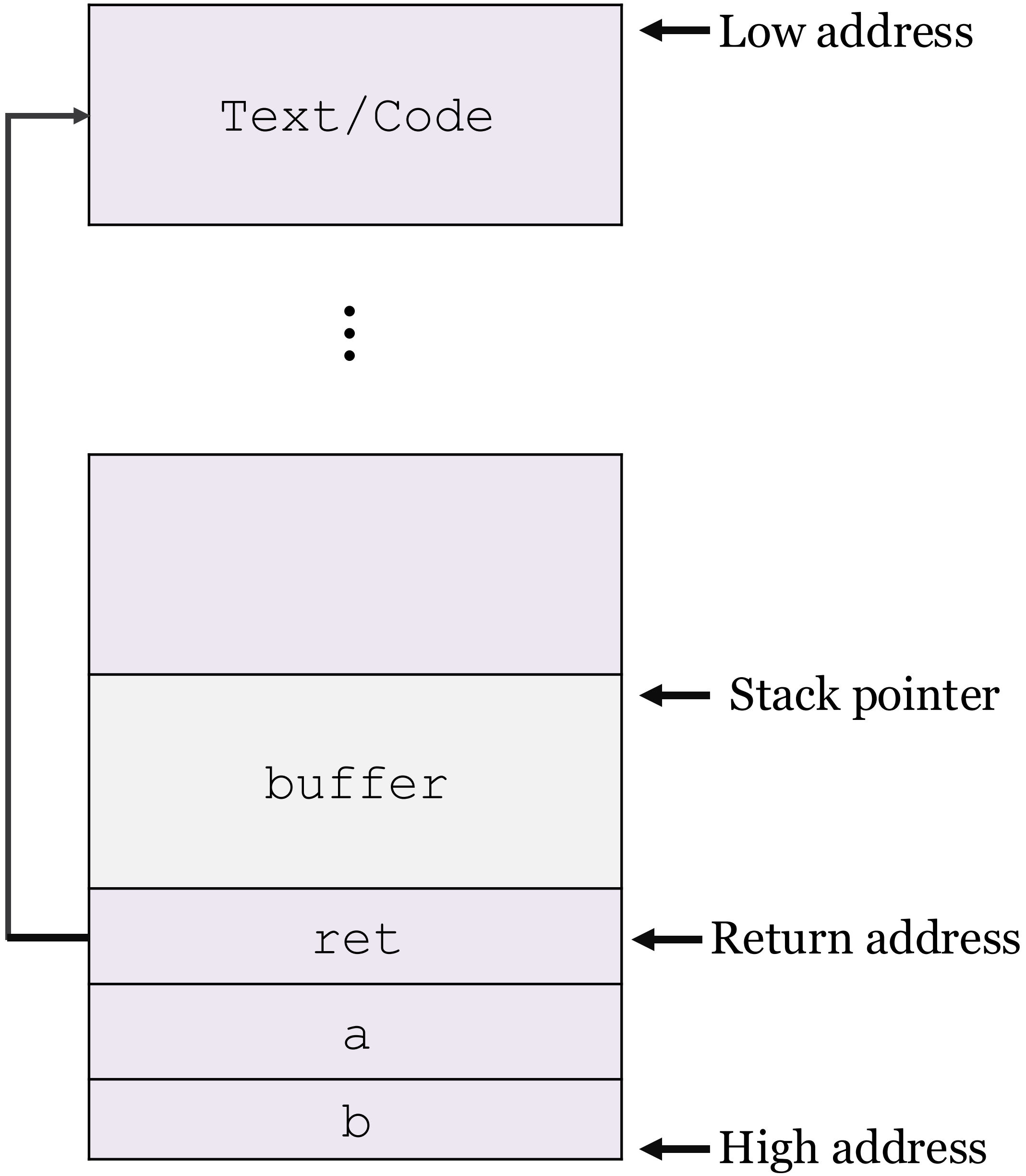
```
void func(int a, int b) {
        int buffer[10];
        buffer[20] = 37;
}
void main(){
        func(1,4);
}
```



Stack smashing (gets)

```
C:> billingprog

Enter Serial Number
432dkgewdk
```

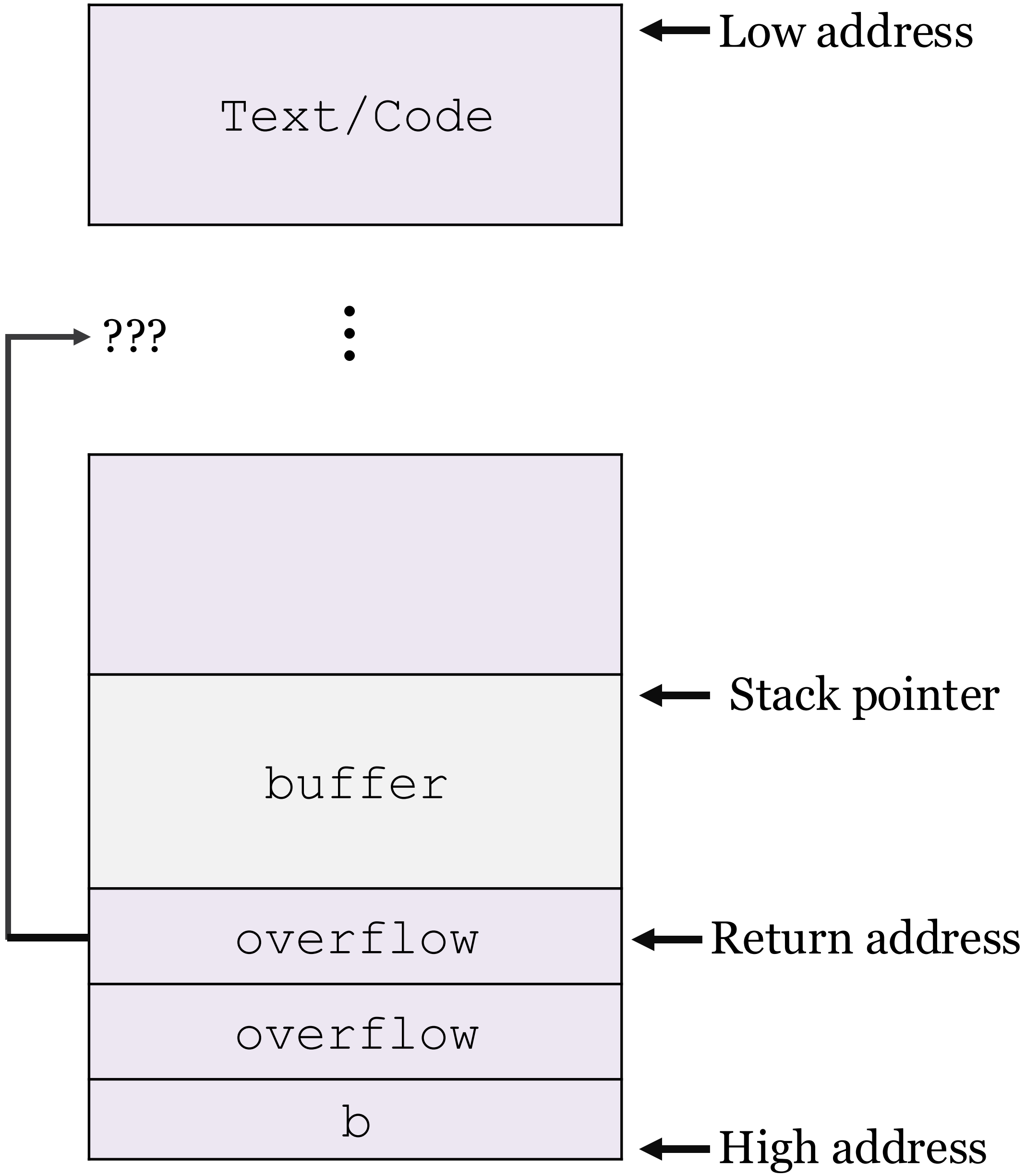
```c
#include <stdio.h>
void func(int a, int b) {
        char buffer[10];
        gets(buffer);
        printf("%s", buffer);
}
void main(){
        func(1,4);
}
```



Stack smashing (gets)

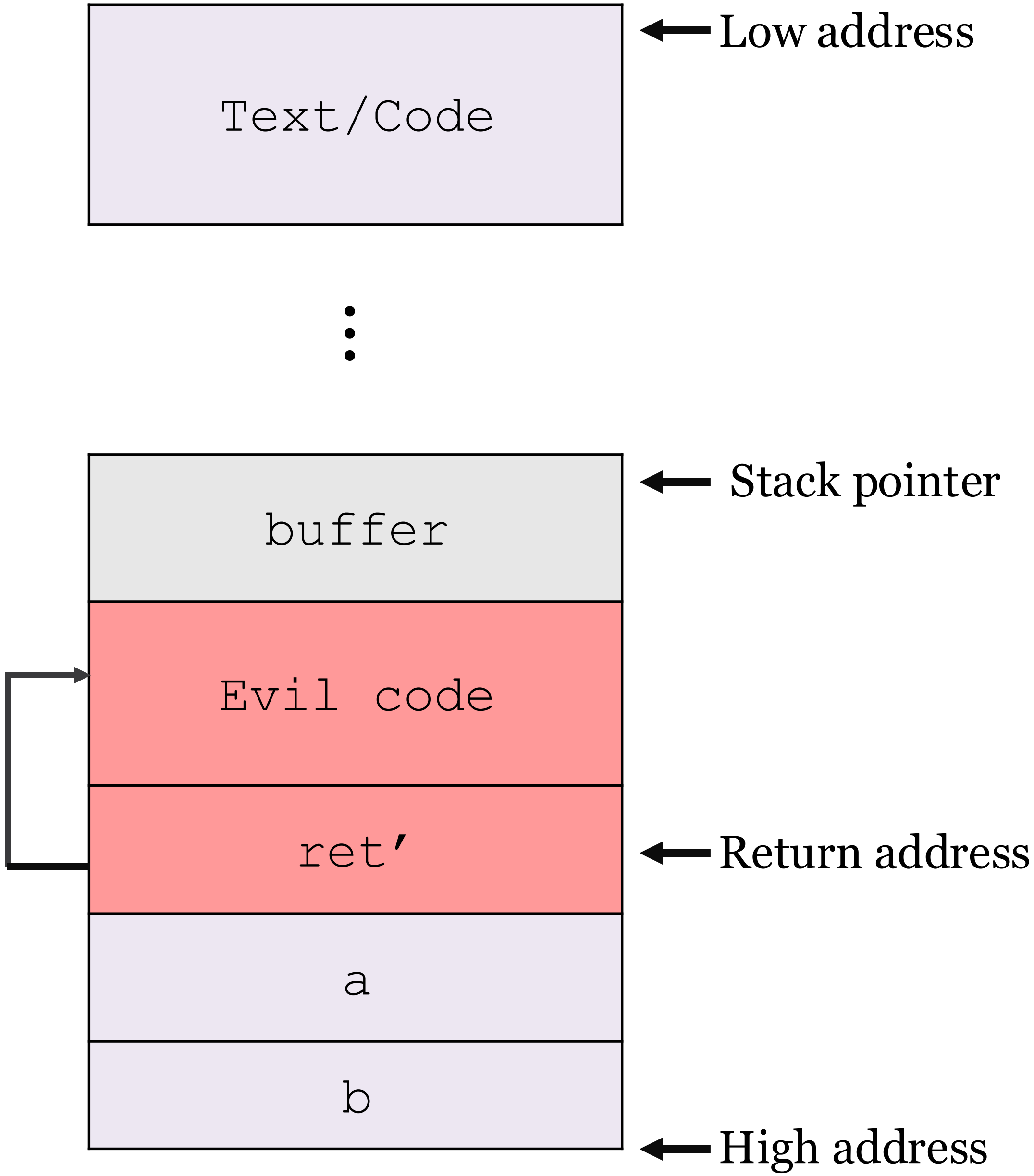
```
C:> billingprog

Enter Serial Number
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaa
```

```c
#include <stdio.h>
void func(int a, int b) {
        char buffer[10];
        gets(buffer);
        printf("%s", buffer);
}
void main(){
        func(1,4);
}
```



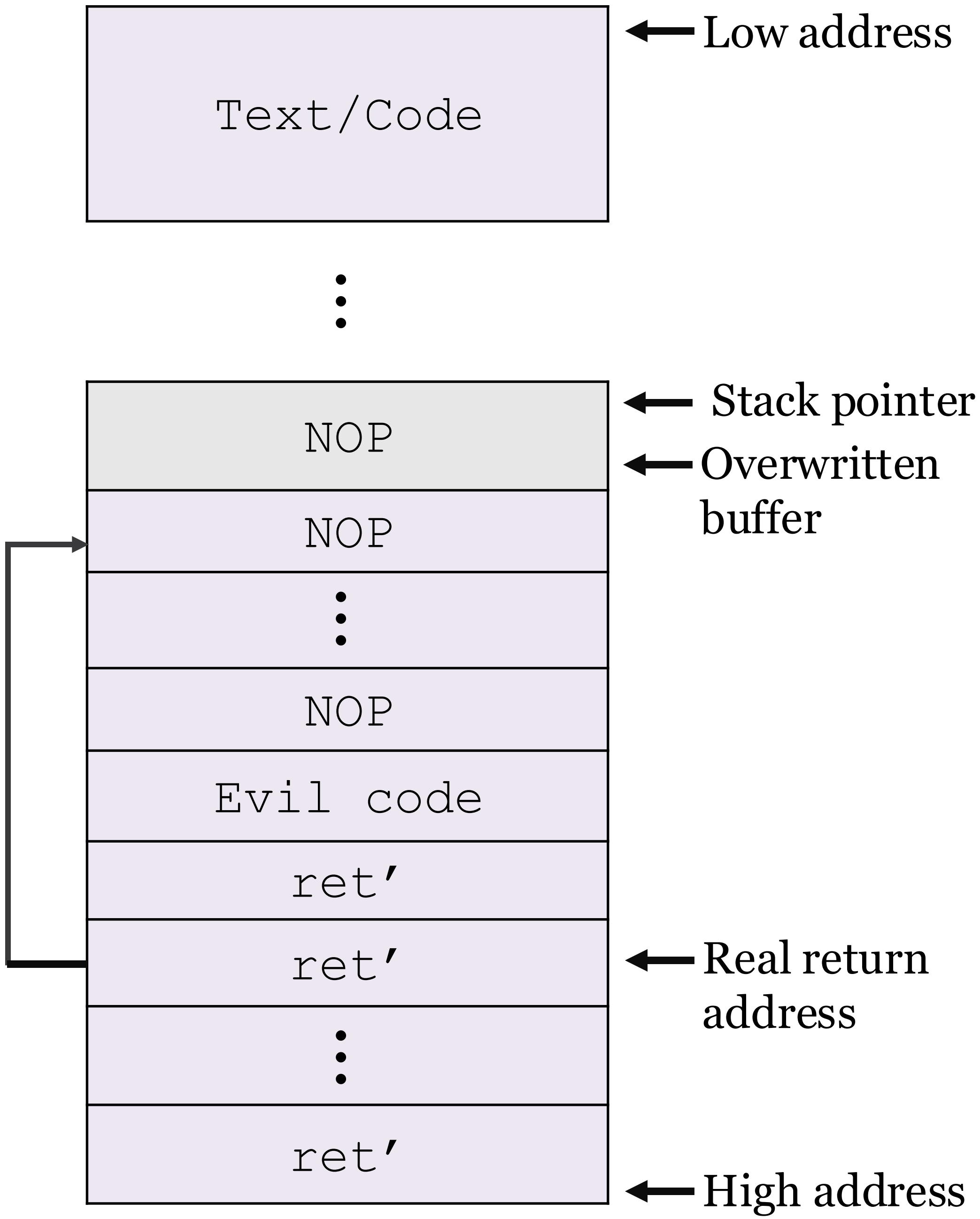
Stack smashing

- Goal: run evil code
- Use buffer overflow to inject evil code onto the stack itself
- Overwrite return pointer so instead of returning after the function



Stack smashing

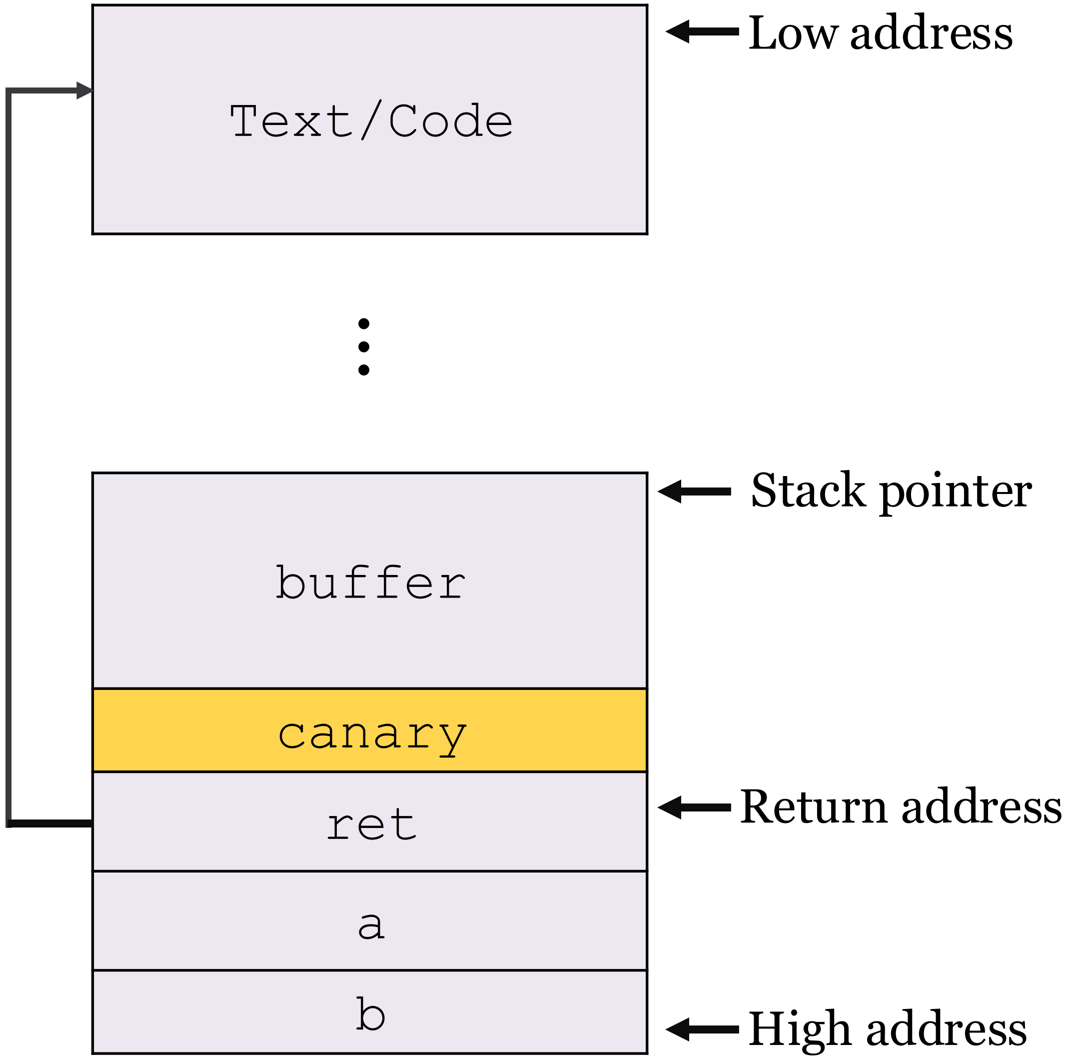
- Attacker's Problem: predicting the exact address of evil code and the return pointer is hard
- Solution: bigger targets
  - Write the return address many times to be sure it is located where the return address pointer points
  - Use NOP's so ret' can point to evil code even if the attacker isn't 100% sure of their relative addresses



Canary

- Add a fixed value right below (lower address) the return address that can be checked before jumping to the return address

```
void func(int a, int b){
        char buffer[10];
}
void main(){
        func(1,2);
}
```



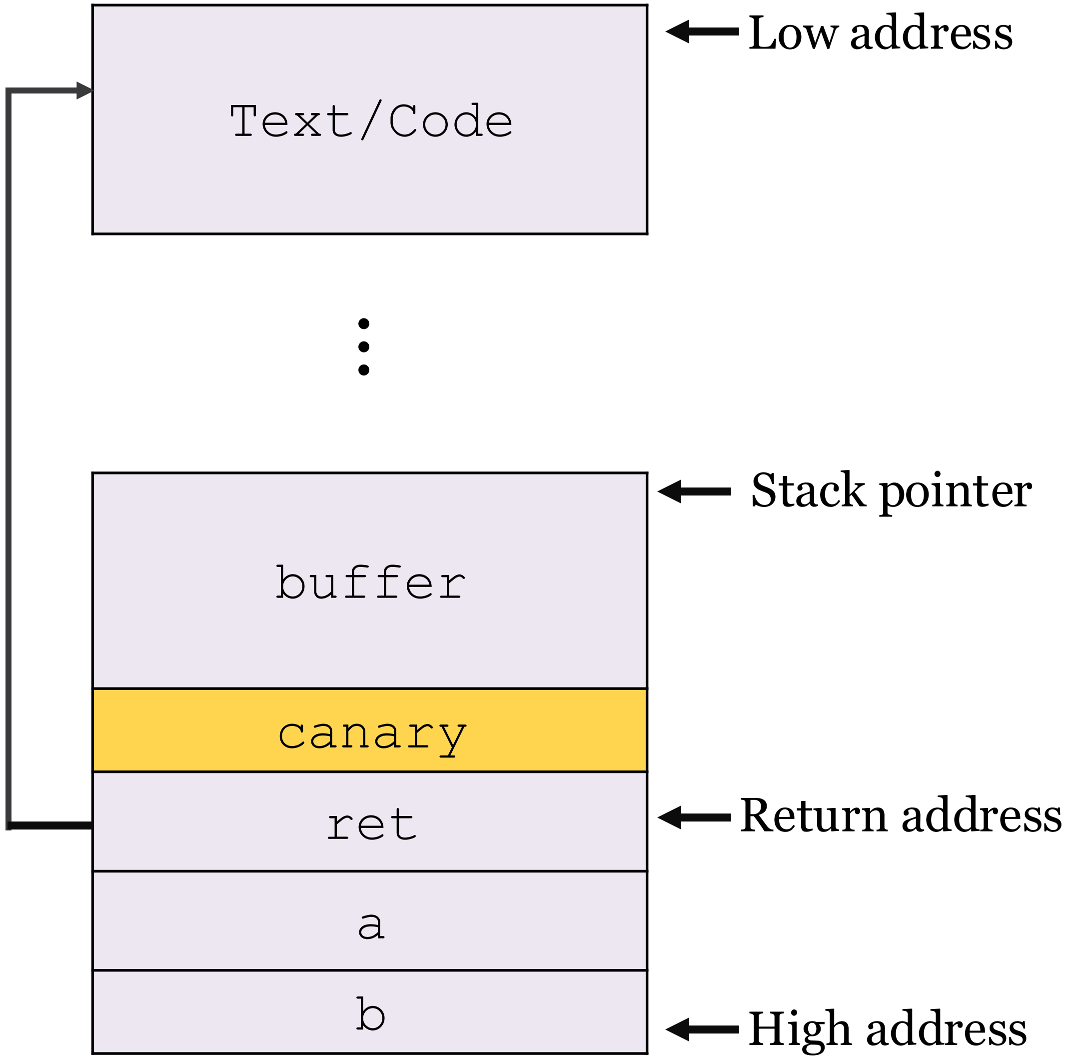
Canary in a Coal Mine

- Coal miners used to bring canaries with them into mines
- Canaries have to breath more often than people so they react quicker to poisonous gas, like carbon monoxide
- If the canary started doing poorly, miners would immediately leave

Smithsonian Magazine. George McCaa, U.S. Bureau of Mines

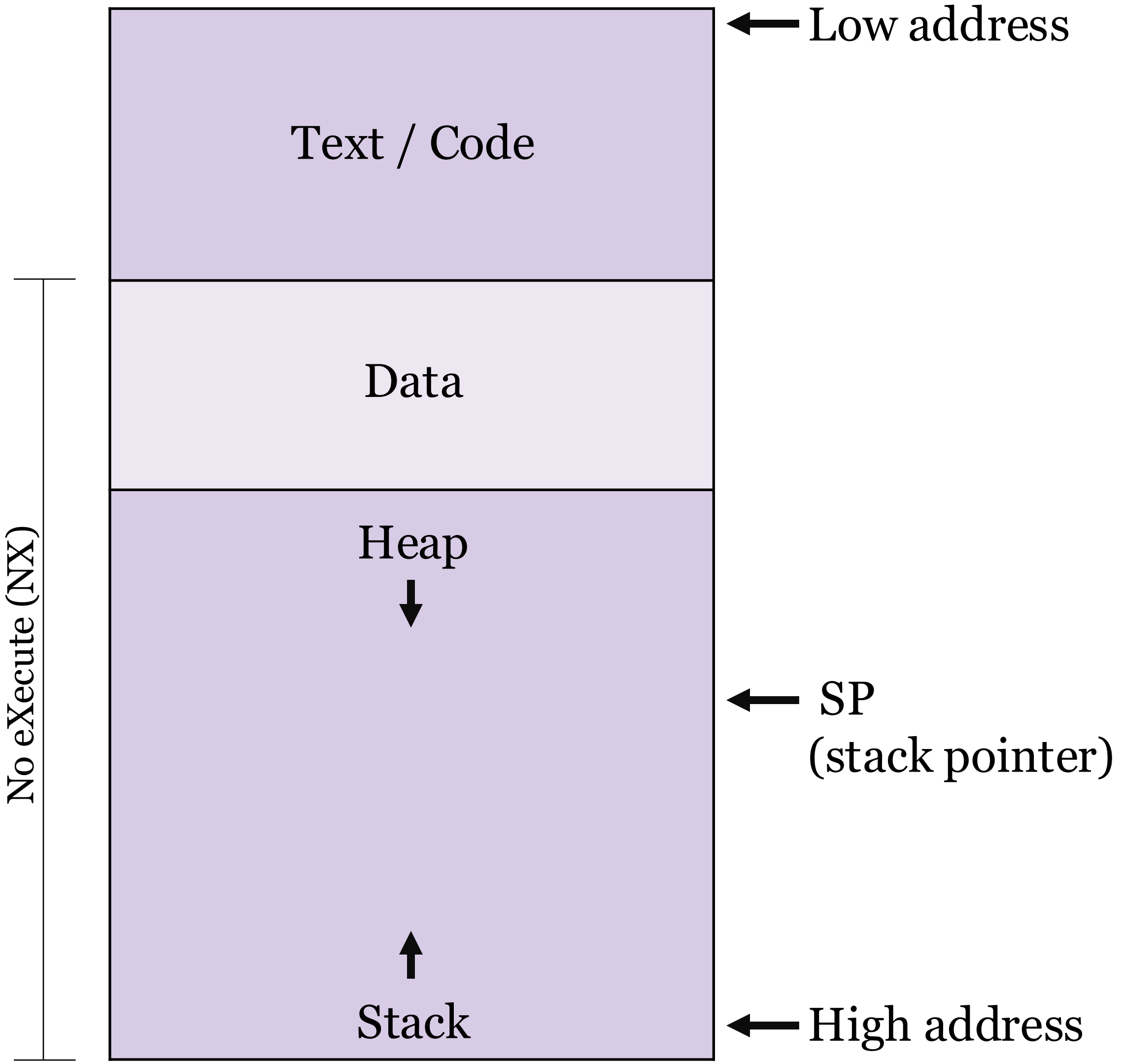
Canary

- Canary sits right above the return address, so will be overwritten by a buffer overflow
- Sneaky overflow could just write canary value back the way it was
- `0x000aff0d` is the most common canary value to use
    - `0x00` is the string termination. Any buffer overflow string containing it will terminate and not overwrite the return address



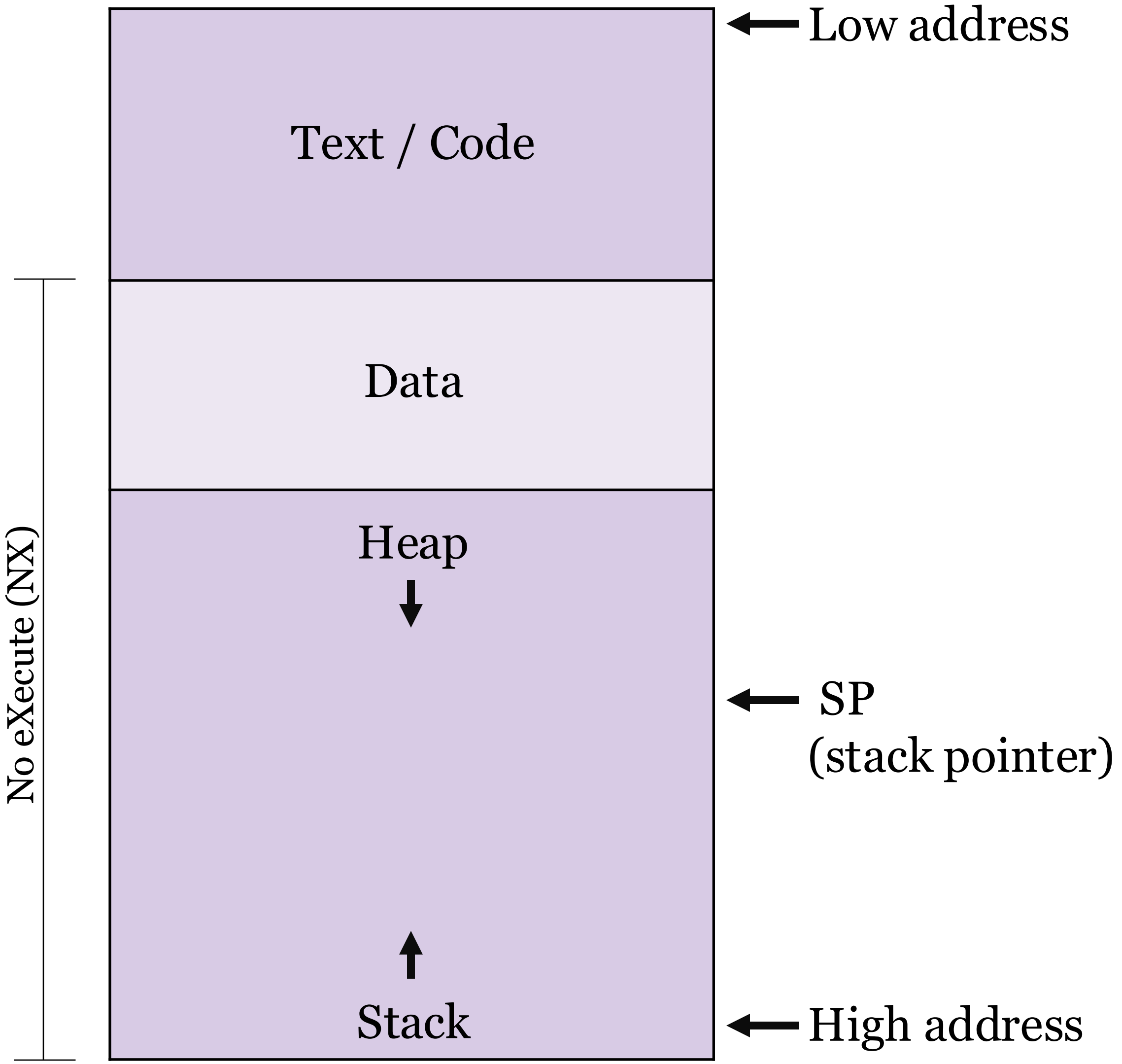
NX bits: Access control preventing execution on stack

- Hardware supports no execute (NX) bits
- Memory can be flagged as NX, so code sitting in that memory cannot execute
- Simple and effective
- But … only applies to code on the stack



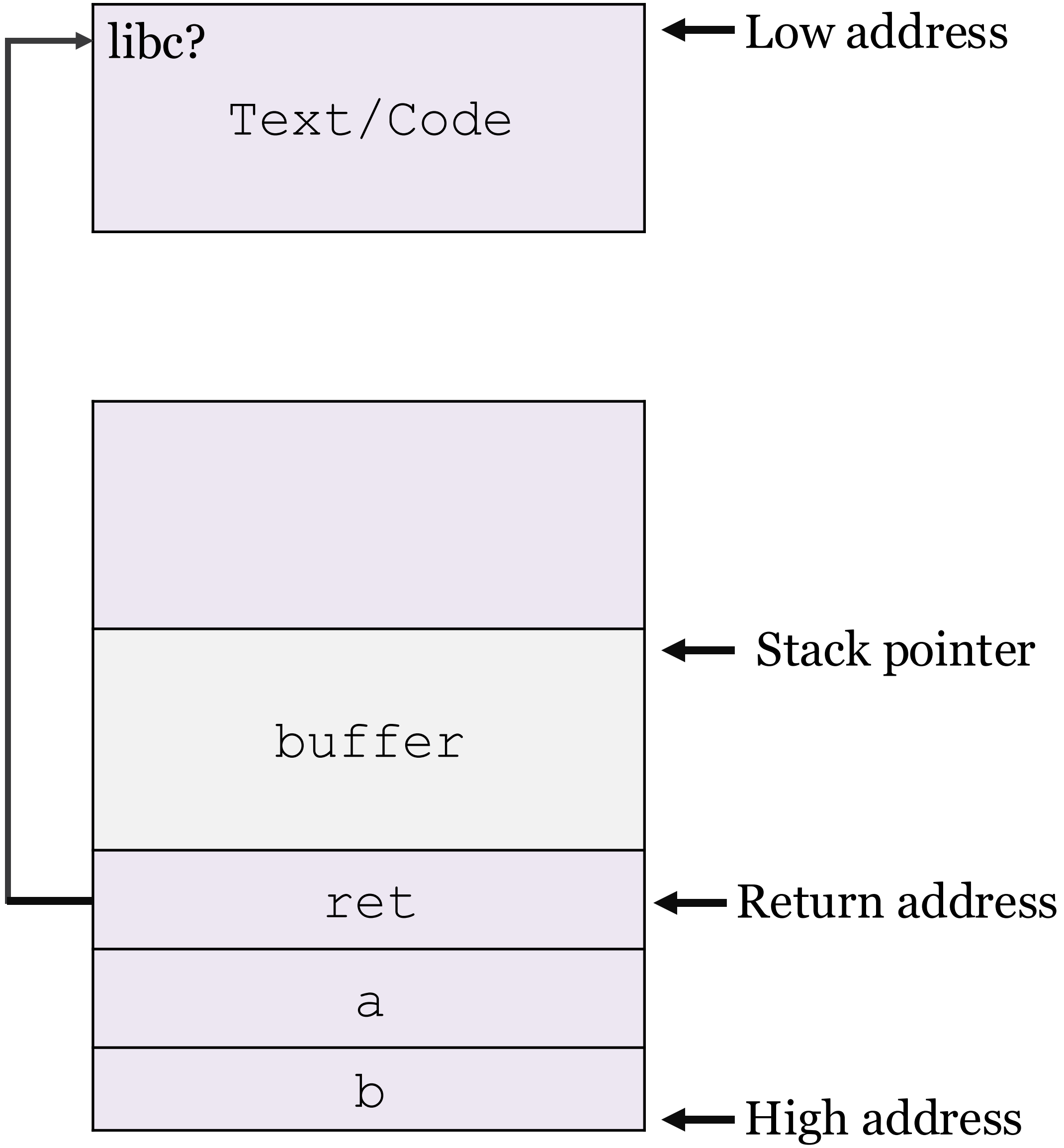
Return to libc attack

- Circumvent NX bit limitations
- Overwrite return pointer with an address in the libc library
- Library is in Text/Code, so no NX bit
- Effectively create evil code out of calls to libc
- Or even calls that jump into the middle of libc functions



ASLR: Address Space Layout Randomization

- Stack smashing works because attacker can predict the address of code
- But there is no reason addressing needs to be linear, the OS can adjust at runtime
- ASLR randomizes the location of key data areas of a process such as: executable code, and positions of stack, heap, and libraries.



INCOMPLETE MEDIATION

Incomplete Mediation

- Buffer overflows are an example of "incomplete mediation"
- Mediation – checking
- Incomplete mediation – failing to check the authorization and properties of a subject/object before using it

Client sends:
https://exampleShop.com?price=4.99&user=4837&login=true

Server:

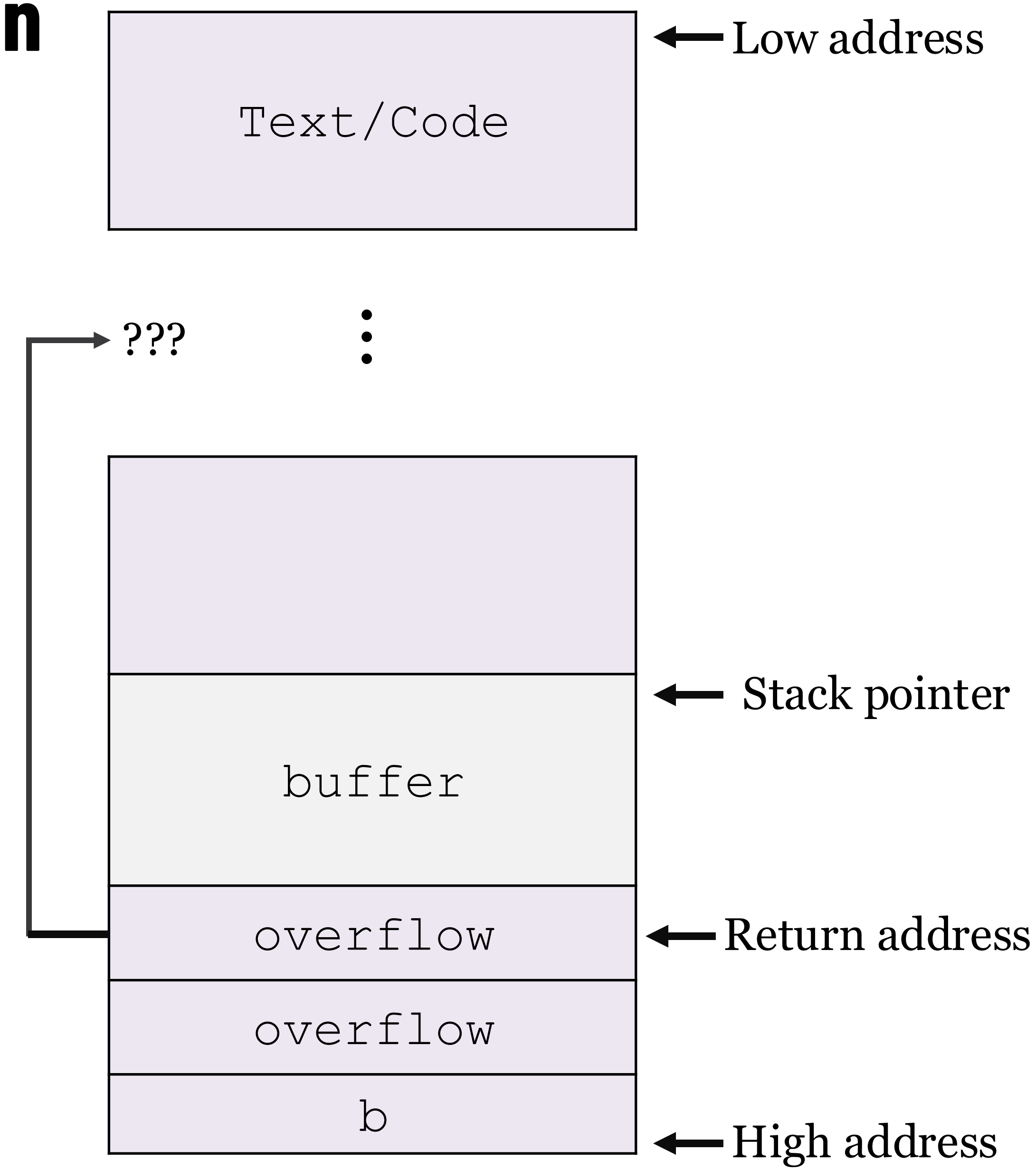
```
function f(price, user, login)
        if(login = true)
                chargeUser(price, user)
```

No server-side check if user is logged in. No server-side check if price is appropriate.

Stack smashing – incomplete mediation

- Below code never checks the length of the buffer
- Worse, gets doesn’t even have a parameter to state what length of string is expected

```c
#include <stdio.h>
void func(int a, int b) {
        char buffer[10];
        gets(buffer);
        printf("%s", buffer);
}
void main(){
        func(1,4);
}
```



“A system which is unspecified can never be wrong, it can only be surprising.”

Flaws

1. Earn or give but never assume trust
2. Use an authentication mechanism that cannot be bypassed or tampered with
5. Define an approach that ensures all data are explicitly validated

Client sends:
https://exampleShop.com?price=4.99&user=4837&login=true

Server:
```
function f(price, user, login)
        if(login = true)
                chargeUser(price, user)
```

No server-side check if user is logged in. No server-side check if price is appropriate.

Thinking outside the bounds

- A key to “hacking” is asking “what if….”
- Look at the example below, the parameters are clearly expecting certain values, what would happen if you gave different values?
- What if for a date you submitted 1800Jan40?

Client sends:
https://exampleShop.com?price=4.99&user=4837&login=true&date=2024Jan17

Server:

```
function f(price, user, login)
        if(login = true)
                chargeUser(price, user)
```

Questions