Secure Programming

CS3524 Distributed Systems

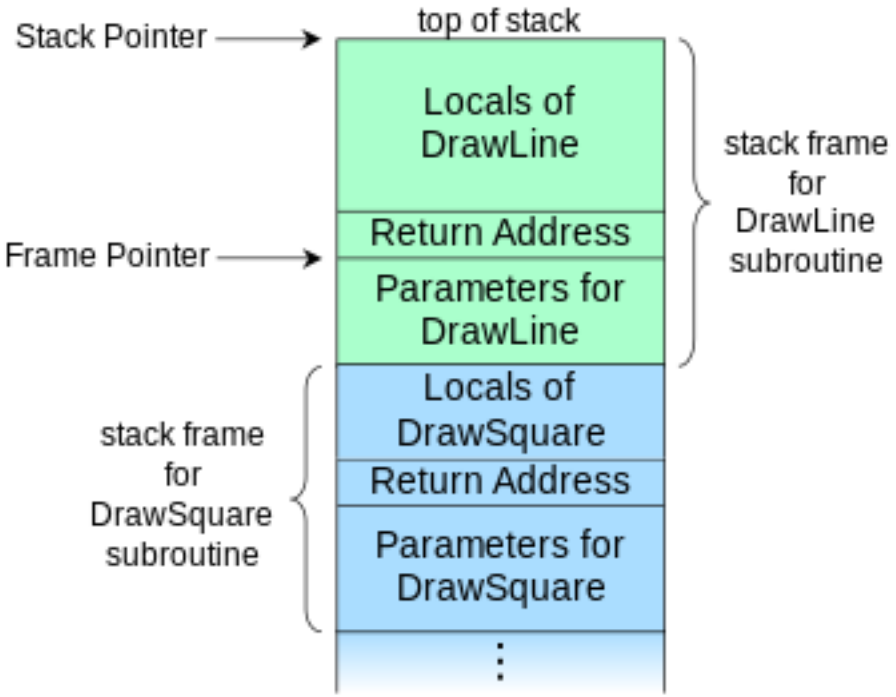
Lecture 15

Secure Programming

- Writing code that is difficult to attack
  - Free of dangerous bugs (from security perspective)
  - General principles
  - Language-specific rules: C, Java, HTML, …
- Taken very seriously by vendors

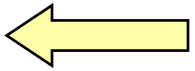
Buffer Overflow

- Problem
  - Program copies a string inputted from the user into a fixed-length buffer on the stack
  - User can supply longer-than-expected string which overruns the buffer
  - Can manipulate the stack frame of a method call – overwrite return address on stack
- This is mostly a problem in C/C++
  - It becomes a Java issue when library code written in C/C++ is called



Example

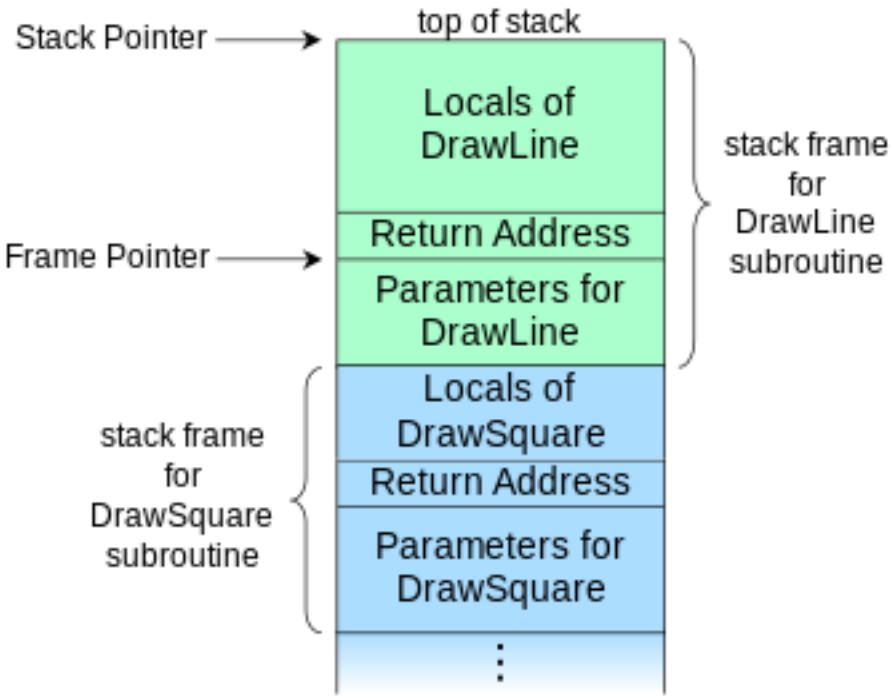
```
int main(int argc, char* argv[]) {
    silly(argv[1]);
    return 0;
}

void silly(char* input) {
    char buf[10];
    strcpy(buf, input);
}
```



Strings of 10 or more characters will cause an overflow

Why is this a problem?

- C's handling of strings:
  - `strcpy` doesn't check the length of the string being copied
  - The stack holds both control info (return address) and local variables
- Can also cause overruns in arrays (which are allocated on the heap)
  - Java error message: "Array index out of bounds"



Size-limited Copying

- C has functions that include a length, but they can be tricky to use correctly
  - `strncpy` – limits copying to B bytes
  - Problem: How many bytes needed to hold a string of N characters?
    - UNICODE – each char may need 2 bytes
    - C strings always need an extra byte at the end (for '\0')

Solution to the example

```
int main(int argc, char* argv[]) {
    notSoSilly(argv[1]);
    return 0;
}

void notSoSilly(char* input) {
    char buf[10];
    strncpy(buf, input, sizeof(buffer));
       buf[sizeof(buffer) - 1] = '\0';
}
```

C++ Strings

- C++ has proper String structures
  - Like Java
- Standard template library, Microsoft Foundation Classes, etc.
- Much safer!
- Why doesn't everyone use C++ String classes?
  - Legacy code and code fragments
  - Programmer habit
  - Concerns about speed, portability
- Changing the way programs are written is a slow business …

Biggest Source of Attacks

- Problem behind 50 – 75% of attacks on browsers, servers, operating systems, etc.
- Including attacks
  - DirectShow, Mozilla bugs that allow web graphic files to take over a browser

Java Security Issues

Java Issues

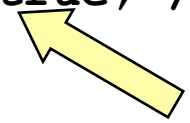
- Exceptions – how well handled?
- Hardcoded passwords, etc
- Inheritance – what does your class really do?
- Mutability – internal objects can be checked by external code
- Dangerous “features” – inner classes, initialisation, cloning, serialisation

Correct Exception Handling

- In general, assume attacker is a **program** calling an API, not a user entering input (as in buffer overflow)
  - Makes Java weaknesses harder to exploit?
- Exceptions
  - Make sure your exception handling leaves the program in a correct state!
    - Printing an error message is not enough
    - Maybe, exception handling leads to a return from such a method or the throwing of another exception

Example

```
boolean checkPIN(String userPIN, int realPIN){
  try {
    int userPINint = Integer.parseInt(userPIN);
        if (userPINint != realPIN)
            return false;
  }
  catch (NumberFormatException e) {
          System.out.println("PIN is not a number!");
  }
  return true; // ???
}
```



Still returns true after NumberFormatException is caught

Hardcoded Passwords

- Hardcoded passwords are a problem
  - E.g:

```
dbURL="jdbc:mysql://localhost/db?user=root&password=secret";
```

- Password (e.g. "secret") can be obtained from compiled code
  - class, war, jar, etc. file
  - Serialised object written to file
  - Dump file, or memory sniffer

Inheritance

- Inheritance
  - If your class inherits from a superclass, what methods does it inherit?
    - Do you even know ?
    - Could these be changed (e.g. in a new version of Java)?

Mutability

- In Object-oriented programming
  - Immutable objects cannot be modified after creation (can be regarded as a “constant”)
  - Mutable objects can be modified after creation – these are the objects we are used to manipulate in our programs
- Mutable objects can be dangerous – an external class can change your private internal data structures
  - if an external class retains a reference to it
  - Copy mutable objects provided by user
  - Copy mutable objects given to the user

Example

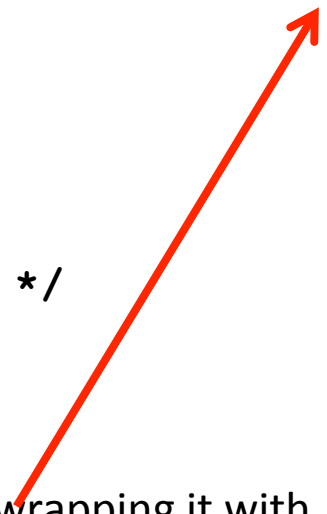
Declaring list as final does not help, still mutable!

```
class Cart {
   private final List items;
   public Cart(List items) {
      this.items = items;
   }
   public List getItems() {
      return items;
   }
   public int total() {
      /* return sum of the prices
*/
   }
 }
```

- An instance of this class is *not* immutable: one can add or remove items either by obtaining the field items by calling getItems() or by retaining a reference to the List object passed when an object of this class is created

Example

```
class ImmutableCart
{
   private final List items;
   public ImmutableCart(List items) {
      this.items = Collections.unmodifiableList( new ArrayList(items));
   }
   public List getItems() {
      return items;
   }
   public int total() {
      /* return sum of the prices */
   }
}
```



- Make list immutable / **read-only** by wrapping it with `Collections.unmodifiableList()`
- We cannot change the List, but single items in this list can be changed:
  - Items in a list are instances of any arbitrary class, these classes may have methods that allow manipulation of the content of such a list item
  - Solution: use Decorator pattern: wrap each item of the list with “decorations” that overwrite any “dangerous” / manipulative methods of such an item
  - (see Decorator pattern: http://en.wikipedia.org/wiki/Decorator_pattern)

Example

- Let's assume, we want users to acquire a software license for using a web service
- The price depends on how long a user wants to use the service
- Attack: A user wants to save money: use online payment system to acquire license for short period, and manipulate transferred license afterwards by changing the expiration date

```
public final class Period
{
  private final Date start;
  private final Date end;

  public Period(Date start, Date end)
  {
    if (start.compareTo(end) > 0) {
      // throw exception
    }
    this.start = start;
    this.end = end;
  }
  public Date start() { return start; }
  public Date end()   { return end; }
}
```

Context of Use

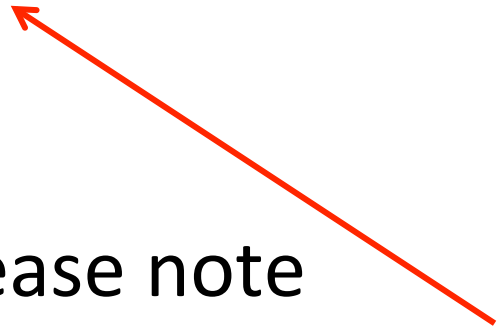
- Let's assume we use the class Period in order to instantiate a software license for this particular time interval

```
getLicense(Date start, Date end, String account)
{
   Period p = new Period(start, end)
   // charge account based on length of period
   // set software so only works in Period P
   . . .
}
```

- Attacker's goal
  - Get a license cheaply (short period)
  - Hack period after license was issued and paid for to extend it without payment
  - (this can also be done with the class java.util.Calendar

Attack 1

```
Date start = new Date(); // current time
Date end   = new Date(); // current time
Period p = new Period(start, end);
end.setYear(78);
```



- Please note
  - Use reference “end” to manipulate end date, after it was used by “Period”

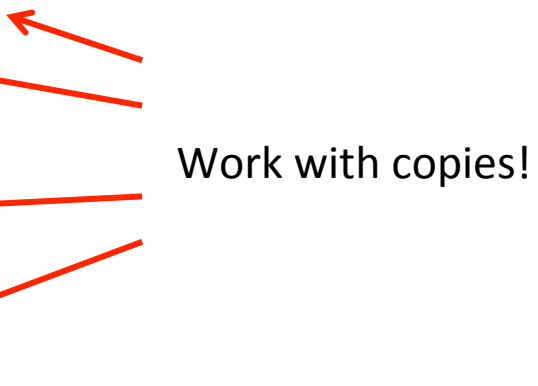
Attack 2

```
Date start = new Date(); // current time
Date end   = new Date(); // current time
Period p = new Period(start, end);
p.end().setYear(78);
```

- Essentially the same attack as we can access references to actual start and end Date objects of period through the methods start() and end().

Secure Version

```
public final class Period
{
  private final Date start;
  private final Date end;

  public Period(Date start, Date end)
  {
    if (start.compareTo(end) > 0) {
      // throw exception
    }
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
  }
  public Date start() {
    return new Date(start.getTime());
  }
  public Date end() {
    return new Date(end.getTime());
  }
}
```



Mutability: Solutions

- Make objects immutable if possible
  - Additionally, immutable objects are thread-safe, hence improve concurrency
- If not possible, make them cloneable and return copies
- If you return objects like Arrays, Vectors, Hashtables, etc., remember that these are mutable, hence their contents can be changed

- Exception:
  - Sometimes it may be necessary to keep sensitive information in mutable data types
    - This allows it to be explicitly cleared at the earliest possible time

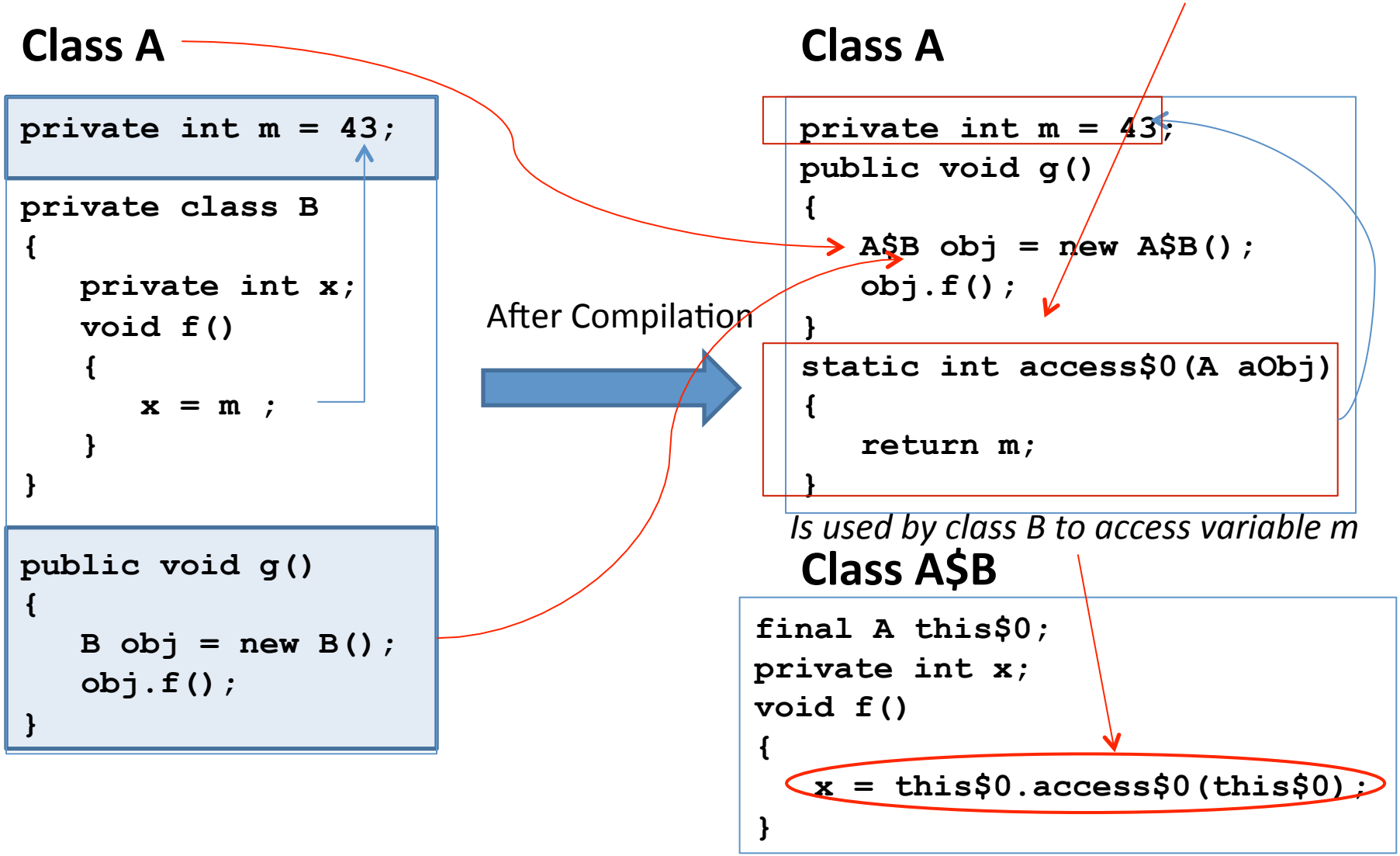
Features to avoid

- Inner classes
  - Wrecks private declarations
  - Initialisations – attacker can bypass those
  - Cloning – allows attacker to bypass constructors
  - Serialisation – attacker can read from disk and find inner state

Problem with inner classes

Access to private variables in enclosing class

Access method is added to allow class B access variable m (which is private to class A)



Problem with Inner Classes

- Class C in same package can now access m!
  - Because we know the naming conventions the compiler uses for creating an access method

Class A

It is possible to retrieve the value of m:

```
private int m = 43;
public void g()
{
    A$B obj = new A$B();
    obj.f();
}
static int access$0(A aObj) {
    return m;
}
```

Undesired Access !!

Class C

```
public fun()
{
    A a = new A();
    . . .

    int x = invokeMethod( "access$0", a );
}
private int invokeMethod ( . . . )
{
    . . . // use Reflection in Java
}
```

Initialization Management

- Make all variables private
- If you want to allow outside code to access variable in an object, this should be done via get/set methods
- This keeps outside code from accessing uninitialised variables
- Add a new private boolean variable, called “initialized”, to the class
- Have the constructor set this variable as its last action before returning
- Have each non-constructor method verify that “initialized” is true, before doing anything

Avoid Cloning

- Make class uncloneable by adding the following method (overrides clone() method):

```
public final Object clone() throws CloneNotSupportedException
{
  throw new CloneNotSupportedException();
}
```

Serialisation Management

- Can make entire class unserialisable by adding the following method

```
private final void writeObject(ObjectOutputStream o)
  throws java.io.IOException
{
   throw new java.io.IOException("Not serializable");
}
```

Serialisation Management

- Can use “transient” modifier for member variables of classes, that are not to be serialised:

```
class UserInfo implements java.io.Serialisable
{
  private String name;
  private transient String password;
...
}
```

- When instances of this class are serialised:
  - Value of “name” is retained
  - Value of “password” will be null

Serialisation Management

- Create your own serialization and deserialization methods

```
private void writeObject(ObjectOutputStream out)
        throws IOException
{
   // Write your own behaviour here.
   // E.g. encrypt sensitive information
}
private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
{
   // Extract (and decrypt) information
}
```