Engineering Secure Software

SECURE DESIGN PATTERNS

Key Security Design Principles

- Today’s design patterns hit upon some key principles
  - Distrust by default
  - Defense in depth
  - Least privilege

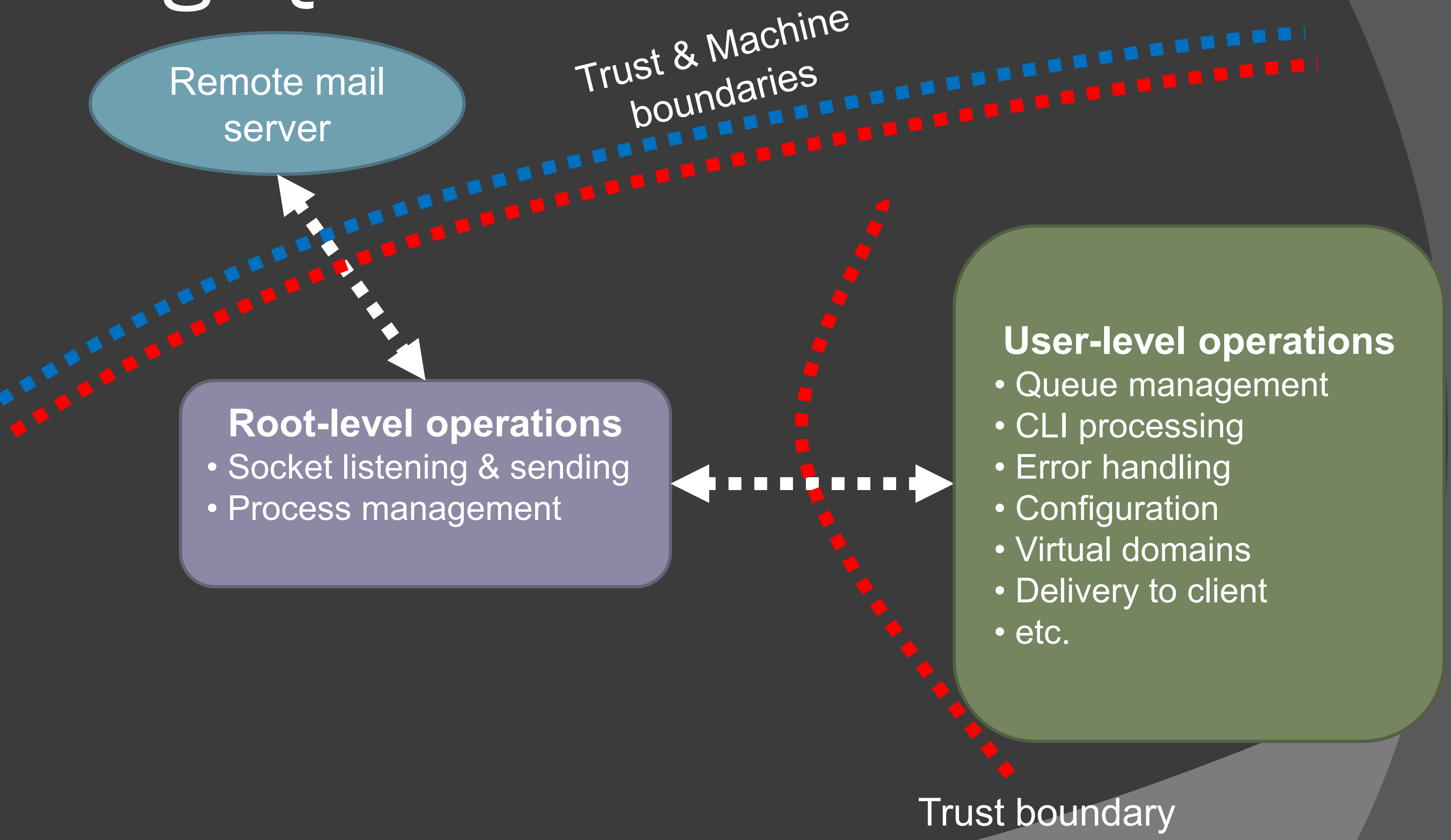
Distrustful Decomposition

- Problem: many programs run with elevated permissions, and need those permissions

- Solution
  - Decompose the system into separate processes with separate permissions (i.e. `fork()`)
  - Communicate via pipes, domain sockets, or files
  - Each process distrusts the other
    - e.g. validate the input from the other process
    - e.g. re-check credentials and integrity mechanisms
  - Allows separation of privilege with the different processes running at different permissions levels

- Intent
  - Reduce impact of an exploit
  - Incorporate distrust at the architecture level

e.g. QMail



Remote mail server

Trust & Machine boundaries

Root-level operations

- Socket listening & sending
- Process management

User-level operations

- Queue management
- CLI processing
- Error handling
- Configuration
- Virtual domains
- Delivery to client
- etc.

Trust boundary

Secure Visitor

- Problem: encapsulating an operation across related objects (e.g. hierarchy), but we want authorization

- Solution
  - Visitor pattern, but with credentials
  - The visited objects get to choose their credential level, not the visitor

- Benefits
  - Authorization is done in visited, not the visitors
  - Some visited objects can choose to never be visited

e.g. CIAOrganization Interfaces

```
public interface IVisitable {
   public <T> T accept(IVisitor<T> visitor, Clearance c);
}

public interface IVisitor<T> {
   public <T> T visit(Director d);
   public <T> T visit(Manager m);
   public <T> T visit(Technician t);
   public <T> T visit(Spy s);
}

//usage:
// director.accept(new AuditTravelVisitor(), clearance);
```

e.g. CIAOrganization Tree

```
public class Technician implements IVisitable {
   public <T> T accept(IVisitor<T> visitor, Clearance c) {
       return visitor.visit(this);// always visit
   }
}
public class Manager implements IVisitable {
   public <T> T accept(IVisitor<T> visitor, Clearance c) {
       if (c.hasClearance("Secret"))
         return visitor.visit(this);
       else
         throw new SecurityException("Authorization required");
   }
}
public class Spy implements IVisitable {
   public <T> T accept(IVisitor<T> visitor, Clearance c) {
     //never visit
     throw new SecurityException("Not visitable!");
   }
}
```

Input Validation Aspect

- Problem: input validation is needed on beans (i.e. just getters and setters)

- Solution
  - Use aspect-oriented programming to provide input validation on all setters
  - New method? Validation is already called

- Intent
  - With unit testing, forces the developer to come up with the input validation early on
  - Encapsulates input validation in one place, without the rest of the system to remember to use it

e.g. Sales

```
public aspect SalesInputValidator {

  pointcut validate(String arg): execution|*
   Sale.set*(String) && args(arg)


   before (String arg): validate(arg){
      if (!str.matches("[a-zA-Z]*"))
            throw new IllegalArgumentException("Input
                  not valid");
   }
}

 sale.setProduct(“123“); //exception is thrown here
```

Secure Logger

- Problem: sensitive logs are piped to stdout, or other insecure means

- Solution
  - Pipe logging statements via SSL to a separate server
  - Provide more performance-intensive filters for a more organized log

- Benefits
  - Fast operation once the sockets are setup
  - Compromising the logger or server doesn’t compromise both
  - Offline analysis is easier