Secure Software Design

Andey Robins

Spring 23 - Week 6

Secure Design Patterns

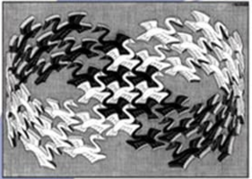
Outline

- Final Project Introduction
- Why Design Patterns
- Security Patterns
- Security Anti-Patterns

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Why Design Patterns?

- A layer of abstraction making design straightforward
- A simplification of some designs
- A shared language

Example: Subscriber Pattern

Consider, you want to be able to register an event listener that can have information pushed out to it when something changes. For instance, you want to register an email service that will send out an email whenever a YouTube channel you're subscribed to publishes a new video.

```
let subscriptions = [
    'Code Bullet',
    'vlogbrothers',
    'VSauce 2'
];

const checkForUpdates = () => {
    for (sub in subscriptions) {
        if (checkForUpload(sub)) {
            notify();
        }
    }
}

registerHourlyJob(checkForUpdates);
```

```
let subscribers = [];

const registerSubscriber = (condition, callback) => {
    subscribers.append(() =>
        condition() ? callback() : (() => {return})
    );
}

const notifySubscriberss = () => {
    for (subscriberCallback in subscribers) {
        subscriberCallback();
    }
}
```

```
const upload = (videoFile) => {
    saveFileToChannel(videoFile);
    notifyListeners();
}

const subscribe = (notificationPreference) => {
    registerSubscriberCallback(
        () => notificationPreference(),
        emailUserCallback
    );
}
```

Dogmatic Design Patterns

Some developers like to adhere very strictly to design patterns, sometimes going so far as to require a pattern for any changes. This is problematic:

- Forces higher level abstractions
- Trends towards over-engineered code
- Obscure implementation details

When Should You Use Design Patterns

1. When you’re trying to create a feature that has a clearly applicable pattern
2. When you need to generalize functionality
3. If your design needs higher levels of abstraction
4. When using a pattern makes it easier to discuss the design

Secure Design Patterns

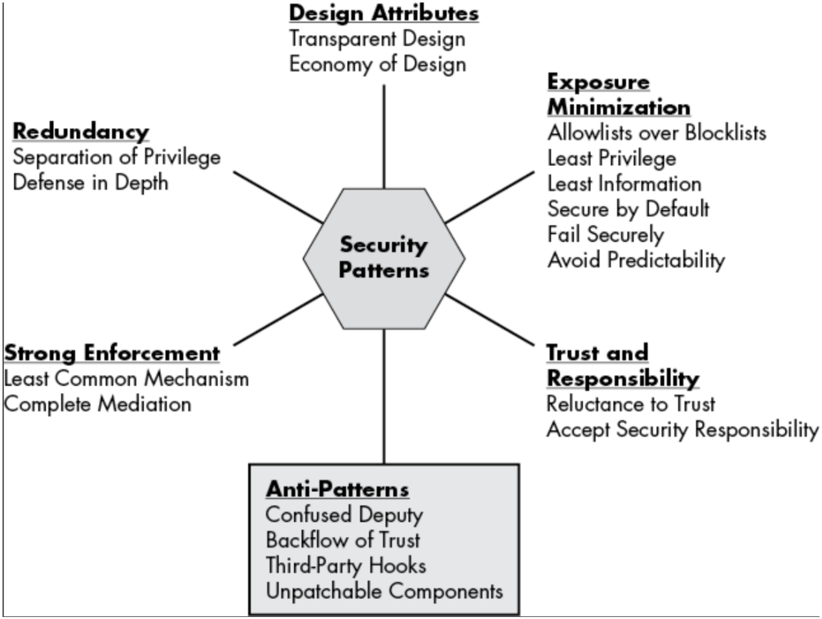


Figure 2: Overview of Secure Design Patterns

Design Attributes

What does secure design look like? *Simple and Transparent*

- Transparent Design
- Economy of Design

Economy of Design

Simpler designs have fewer bugs, which means fewer vulnerabilities.

Consider LEGO vs Model Kits:

- Individual components are either generic or special built
- Combinations are either infinite or individually mandated
- Purpose is clear or contextually dependant

Transparent Design

Secure protection should never rely on secrecy

Do the leg work to build a secure design whether or not the details are ever made public.

Figure 3: What did disclosure of this exhaust port cause?

Secrets

This is not to say nothing should ever be kept secret. Publishing your design is likely not the best movie for a closed-source project; however, the disclosure of the design should not compromise its security.

The opposite of this, *security through obscurity* is an anti-pattern discussed later.

Exposure Minimization

Play it safe unless there is a reason not to

- Allowlists over Blocklists
- Least Privilege
- Least Information
- Secure by Default
- Fail Securely
- Avoid Predictability

Allowlists vs Blocklists

Allowlists enumerate what options are safe and are inherently finite.

Blocklists enumerate what options are *unsafe* and are inherently infinite.

- Firwalls
  - It would be nearly impossible to enumerate every single disallowed IP
- Password input characters
  - You can't enumerate every possible unicode character to block, so pick some allowed ones
- Sudoers

Least Privilege

Only use just enough privilege for the job

- Never clean a loaded firearm
- Unplug power tools when changing blades
- Don't run every command as `sudo`
- "Break in case of fire" glass

Make every effort to lower the privilege needed for an action. When an exploit occurs, isn't it better to have the attacker in a minimal privilege environment than a root privilege one just because it was easier not to design in the trust boundaries?

Least Information

Use the minimum amount of information needed for the job.

The data privacy version of least privilege. This principle can sneak in as time goes on due to re-using old data structures and method calls and only consuming the necessary information while requesting lots of it.

This is all about minimizing the flow of private/sensitive information

```
// from the impressive @sdgfsdh on SO
// get a password from the cli replacing
// input with **** to hide it
const readPassIn = (query: string): Promise<string> => {
    return new Promise((resolve, _) => {
        const rl = readline.createInterface({
            input: process.stdin,
            output: process.stdout,
        });
        const stdin = process.openStdin();
        // -- SNIP password replacement code -- //
        rl.question(query, (value) => {
            resolve(value);
        });
    });
}
```

```
process.stdin.on("data", (char) => {
  let str: string = char + "";
  switch (str) {
    case "\n":
    case "\r":
    case "\u0004":
      stdin.pause();
      break;
    default:
      readline.clearLine(process.stdout, 0);
      readline.cursorTo(process.stdout, 0);
      process.stdout.write(
        query + Array(rl.line.length + 1).join("*")
      );
      break;
  }
});
```

Secure by Default

Software should be secure "out of the box."

Inaction on the part of the principal should not compromise their security.

- Have secure default passwords
- Assume connections use HTTPS and only fallback to HTTP
- Don't allow for configuring insecurely

Fail Securely

[When] a problem occurs, be sure to end up in a secure state.

A fuse physically prevents the state where excess current is flowing through a circuit. Transactions can be our software “fuse.” Robust error handling is another option.

Image Upload

All three can throw an exception: validateBlob, savePhotoBlob, and addBlobToProfile.

```
const onUpload() = (uploadBlob) => {
    let responseCode = 200;
    let url;
    try {
        validateBlob(uploadBlob);
        url = savePhotoBlob(uploadBlob);
        addPhotoToProfile(url);
    } catch (exception) {
        delete uploadBlob;
        url ? deleteFile(url) : continue;
        responseCode = 500;
    } finally {
        return responseCode;
    }
}
```

Avoid Predictability

Any data or behavior that is predicatable cannot be assured to remain hidden.

Consider the example of sequential user IDs from week 2.

Threats caused by predicatability sit in the side-channel space and so cannot truly be totally eliminated, but making use of secure randomness can alleviate many problems.

Instead of this:

```sql
CREATE TABLE Users (
    uid INT AUTOINCREMENT PRIMARY KEY,
    email TEXT NOT NULL,
    bio TEXT
);
```

Do this:

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE Users (
    uid UUID NOT NULL PRIMARY KEY
        DEFAULT generate_uuid_v1(),
    email TEXT NOT NULL,
    bio TEXT
);
```

Trust and Responsibility

- Reluctance to Trust
- Accept Security Responsibility

Reluctance to Trust

Trust should always be an explicit choice.

Cookies are a perfect example. It is useful to set client side cookies and request their return by a client, but since there is no obligation for them to, and the client side cookies could be arbitrarily modified, they shouldn't be assumed to be safe and un-modified upon receipt by the back-end.

"This pattern is straightforward and rational, yet can be challenging in practice because people are naturaly trusting and it can feel paranoid to withhold trust. - Kohnfelder

Accept Security Responsibility

All software professionals have a clear duty to take responsibility for security.

Decide who validates requests and what guarantees to provide downstream.

Example: Web Input Validation

Place a web validation middleware onto requests so that by the time a web request reaches an endpoint, you know that it will be a well formed request. You can then bypass the need to check for simple things like authentication tokens or correct verbs and instead check the veracity of the information.

Alternatively, perform checks at both places for a “defense-in-depth” approach.

Strong Enforcement

It is better to design code so that forbidden operations are structurally prevented.

- Least Common Mechanism
- Complete Mediation

Least Common Mechanism

Maintain isolation between independent processes by minimizing shared mechanisms.

Information *bridging* between processes is a threat to the authorization of any work. If one system can influence another (either in the data it gets or the actions it performs), this threatens the security of both.

Example: Kernel Processes

The kernel will isolate resources and items between processes so that they can't interact by default. The data structures and systems used to do this, if they could be modified by userland code, would be a prime target for subverting another process' work.

Complete Mediation

Protect all access paths in the same way.

Data protected by access control policy should only be accessible after having access checked according to said policy.

Example: Tax System

A tax preparation company wants to protect sensitive customer information, so they don't show SSNs to employees (only managers can look it up). The employees can then help prepare tax documents and request the documents from the system to provide them to clients.

Problem: Tax documents include SSNs.

Levels of Mediation Compliance

High: One single path for access (i.e. bottlnecked)

Medium: Multiple routes with identical checks

Low: Different levels of authorization checks depending on route to data

Exploit: iOS Screentime

"I saw that my 8-year-old sister was on her iPhone 6 on iOS 12.4.6 using YouTube past her screen time limit. Turns out, she discovered a bug with screen time in messages that allows the user to use apps that are available in the iMessage App Store."

Redundancy

- Separation of Privilege
- Defense in Depth

Separation of Privilege

Two principals are more trustworthy than one.

Relying on multiple actors to coordinate in order to subvert a system is a very good way of making everyone shape up and behave. If the admins audit normal users, and a separate team audits the admins, everyone has the incentive to behave.

Example: Nuclear Launch

Two keys must be turned simultaneously in terminals 10 feet apart, preventing any one person from turning those keys.

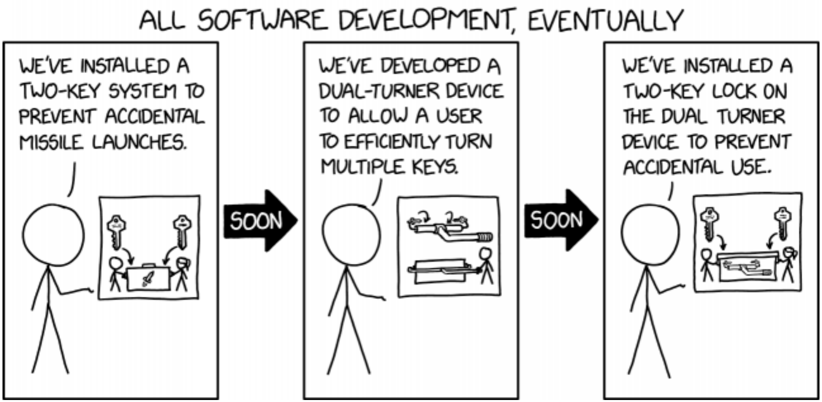

ALL SOFTWARE DEVELOPMENT, EVENTUALLY
WE'VE INSTALLED A TWO-KEY SYSTEM TO PREVENT ACCIDENTAL MISSILE LAUNCHES.
SOON
WE'VE DEVELOPED A DUAL-TURNER DEVICE TO ALLOW A USER TO EFFICIENTLY TURN MULTIPLE KEYS.
SOON
WE'VE INSTALLED A TWO-KEY LOCK ON THE DUAL TURNER DEVICE TO PREVENT ACCIDENTAL USE.

Figure 4: XKCD 2677

Defense in Depth

Combining multiple layers of protection make for a stronger overall defense.

Sandboxing is the practice of putting untrusted code into an untrusted environment where it can run. Often this is also coupled with a pre-check which looks to see if the untrusted code wants to do something like call kernel processes or read memory to be able to stop it from running in the sandbox too if appropriate.

Secure Design Anti-patterns

Anti-patterns

These are not vulnerabilities, but they do carry security risk. Therefore, they are best avoided.

- Confused Deputy
- Backflow of Trust
- Third-Party Hooks
- Unpatchable Components
- Security by Obscurity
- Responsibility Gap

Confused Deputy

1. A judge issues a warrant for the arrest of one someone
2. The deputy receiving the warrant looks up their address and arrests the person living there
3. The person assures the deputy there has been a mistake
4. The deputy isn't hearing it

The twist: The real victim has used a false address to frame the other person

The deputy is acting in accordance with the law and the system as presented, but used their authority wrongly to arrest the wrong person.

This pattern emerges whenever *a callee* causes higher privilege code to execute on the behalf of a lower privileged caller.

In other words, a confused deputy is an anti-pattern when a trust boundary is mistakenly crossed.

Example: Log4j

When modifications to an iPhone name trigger privileged writes to log files, we could say the anti-pattern which emerges is the confused deputy pattern.

Backflow of Trust

Backflow of trust is whenever a lower-trust component controls a higher trust component.

An example is when a sys-admin uses their home computer to manage the enterprise system.

Third-Party Hooks

Imagine we have contracted an AI company to provide monitoring of your business system for our own gain. Due to the AI system requiring constant maintanence and daily monitoring, it requires a specialized punchout through our Firewall. The company responsible for managing the AI system could forseeably exfiltrate any company secrets through this channel with no oversight due to the setup.

Third parties should be trusted as little as possible in accordance with our other principles (i.e. separation of privilege and least common mechanism).

Unpatchable Components

As time goes to infinity, the number of vulnerabilities in code will rise. If there exists no means to patch a system, it is a vulnerability waiting to happen.

Ensure a way to patch any piece of software. Plan for ways to patch critical systems that are otherwise unpatchable.

Security by Obscurity

The security of a system should not rely on the secrecy of its design.

This pattern is to try to hide things by keeping them a secret. For instance, encrypting information with DES and relying on that secrecy to prevent a data leak.

Responsibility Gap

Two principals claiming "I thought you were handling security" while neither does

Operate on a system where both people assume the mantle for security. This leads to:

1. Defense in Depth
2. Redundancy
3. Acceptance of Security Responsibility

Questions?

Next Time

We talk about the “Traditional Design Patterns” and showcase some design pattern code.

Logging Assignment Retrospective

Languages

Language	Repos
Javascript	9
Go	6
Rust	3

Common Feedback

1. Make use of logging library
2. Handle errors
3. Add timestamps to logs

Logging in Javascript

```javascript
// initialize once
export const logger = new Log4js.getLogger("deaddrop");

// log anywhere
logger.info(user, "checked their messages");
logger.error("failed to decrypt database");
logger.warn(user, "failed authentication");
```

Logging in Go

```go
// initialized once
file, err := os.OpenFile("deaddrop.log",
    os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0666)
check(err)
log.SetOutput(file)

// write anywhere
log.Println("INFO", user, "checked their messages")
log.Println("ERROR", "failed to decrypt database")
log.Println("WARN", user, "failed to authenticate")
```

Logging in Rust

```rust
// initialize once
log4rs::init_config(config)
    .expect("expected log config to exist")

// write anywhere
info!("{} checked their messages", user);
error!("failed to decrypt database")
warn!("{} failed authentication", user);
```

Error Handling Advice

1. Don’t treat all errors the same
2. Leverage your type system
3. Refactor when appropriate

Code Examples

Lets Talk Middleware

```
import jwt from "jsonwebtoken";
import fs from "fs";

const key = fs.readSync("key.pem");

const authMiddleware = (req, res, next) => {
  // extract the "Bearer $" from the auth header
  const token = req.headers["Authorization"].split(7);
  if (!jwt.verify(token, key)) {
    res.status = 403;
    res.fail();
  }
  return next();
};
```

Stringing it Together

```
import express from "express";

const app = express();
app.use(authMiddleware);
app.get("/", (req, res) => {
  res.send("Hello World");
});
```

Under the Hood

```
const onHttpRequestReceived = (req, res) => {
  app.getFirstMiddleware();
  if (req.route == "/") {
    res.send("Hello World");
  }
};
```

Multiple Middleware

```javascript
const infoLoggingMiddleware = (req, res, next) => {
  console.log(`endpoint ${req.route} hit`);
  return next();
};

const app = express();
app.use(authMiddleware);
app.use(infoLoggingMiddleware);
app.get("/", (req, res) => {
  res.send("Hello World");
});
```

The Pathway of Composition

```
// . is the composition opperator
/* app -> authMiddleware . infoLoggingMiddleware . (req, re
    res.send("Hello World")
} */
```

Redundant Validation

```
app.use(authMiddleware);
app.get("/", (req, res) => {
  authMiddleware(req, res, () => {
    res.send("Hello World");
  });
});
```

Multiple Redundant Validation

```
app.use(authMiddleware);
app.use(infoLoggingMiddleware);
app.get("/", (req, res) => {
  authMiddleware(req, res, () => {
    infoLoggingMiddleware(req, res, () => {
      res.send("Hello World");
    });
  });
});
```

Do This With “Acceptance of Security Responsibility”

Can be as simple as some additional documentation:

```
const app = express();

// auth middleware required for user authentication
app.use(authMiddleware);
app.use(infoLoggingMiddleware);

/// --- SNIP --- ///

// GET endpoint for root page.
// Only responds with a hello world
// authorization required: authMiddleware
app.get("/", (req, res) => {
  res.send("Hello World");
});
```

Express Secure Defaults

[Refer to this link]

Secure Programming for Web Servers in Go

[A security first project]

See content under
`//go-safeweb/examples/simple-application/server/server.go`
and look for `postNotesHandler`