Secure Software Design

Andey Robins

Spring 23 - Week 3

The World of Threats

Outline

- Threat Modeling
- Attack Surfaces
- Risk
- Mitigation and Prevention
- Example Threat Modeling

Threat Modeling

"Understanding the potential threats to a system is the essential starting point in order to bake solid defenses and mitigations into your software designs." - Loren Kohnfelder

An Extension of "Think like an adversary"

- Specific details and techniques are unimportant
- Focus on high level questions
  - What would they be after?
  - What would we think is important?
  - How do these things differ?
- They only need to succeed once

The Four Questions

- What are we working on?
- What can go wrong?
- What are we going to do about it?
- Did we do a good job?

What Are We Working On?

This is the primary question of software design:

- gather requirements
- identify stakeholders
- propose and plan for features

Going Even More Specific

- What components are we using?
- How do these components interact with each other?
- By what method do we ensure interaction?

Example: TCG-client

Consider that we are writing a client application for a trading card game. We are on the team tasked with developing a system for writing the system for loading players into a game, and you are specifically tasked with handling deck initialization.

Problem: players provide us with their decks and we then tell them to shuffle those decks. How do we prevent "stacked decks?"

What Are We Working On?

A deck shuffling tool which requires the deck to be shuffled

What Can Go Wrong?

- A user says they've shuffled when they haven't
- A user shuffles, but poorly
- A user doesn't even provide a valid deck

What Are We Going to Do About It?

In othe words: “how do we mitigate the threat?”

Threat mitigation is discussed in further depth later and in Chapter 3 of the textbook.

Did We Do a Good Job?

This question is a retro-spective question. It's impossible to answer up front, and often may be nearly impossible to answer definitively after the fact too. Did we do a good job or was this part of the system just never the target of an attack? The only situation where we can clearly answer this is when an attack occurs and we have information on this component's role in the attack.

What Can We Do Better?

If the answer to the previous question is “no,” this is the follow up question I propose.

1. Use the previous attack to create a proof of concept “attack”
2. Brainstorm alternative designs
3. Integrate the PoC into a regression test suite while working on fixes
4. Ensure the same attack won’t work a second time ever again

Returning to Our Example

Create some interfaces:

```
interface Deck {
    cards: Card[],
    shuffle(d: Deck): ShuffledDeck,
}

interface ShuffledDeck {
    readonly cards: ReadonlyArray<Card>,
    draw(d: ShuffledDeck): Card,
}
```

Subsequent functions can only accept a `ShuffledDeck` as an argument, and we can design the system so that the only function which returns a `ShuffledDeck` is the `shuffle` function on the `Deck` interface. This addresses our first concern of ensuring that a user has actually shuffled their deck.

Addressing Poor Shuffling

```
interface Deck {
    cards: Card[],
    shuffle(d: Deck): ShuffledDeck,
}

interface ShuffledDeck {
    readonly cards: ReadonlyArray<Card>,
    shuffle(d: ShuffledDeck): ShuffledDeck,
    draw(d: ShuffledDeck): Card,
}
```

We can now design a workflow where a user shuffles their opponent's deck and then the opponent shuffles their own deck. Similarly, they shuffle our deck before we also shuffle it. We now have redundant randomization which can serve to eliminate the concern that you could have a non-randomizing shuffle function. You're incentivized to present a true randomizer because it will ensure that both decks are truly randomized.

Addressing Valid Decks

We've already ensured it! The type based aproach to interface design makes it so that we can only shuffle a deck and a deck must be made of a list of cards.

Attack Surfaces

Attack Surfaces

Attack surfaces are an attacker's first point of entry to a system.

- Internet
- USB ports
- I2C bus
- Config file
- Physical device

Attack Surface Identification

- Identify assets
- Characterize asset access
- Identify trust and trust boundaries
- Model potential threats

Identify Assets

Our ultimate goal is to be able to protect everything, however this is impossible, and therefore it makes sense to somehow prioritize what we protect.

1. Enumerate assets
2. Identify stakeholders
3. Assign value level

Iterative Approach to Security

Kohnfelder identifies a general application of the agile methodologies from the field to security. Perform incremental passes, seeking to improve the overall system each step.

Some Example Assets and Levels

Asset	Value
Customer post DB	Mid
Frontend source code	Low
Backend source code	Mid
Financial backend	Mid
Moderation backend	Low
Subscriber DB	High

The level of value placed on an asset will often vary between stakeholders

Asset	CEO	Advertiser	DB Admin	User
Customer post DB	Mid	High	High	Mid
Frontend source code	Low	Low	Low	High
Backend source code	Low	Low	Mid	Low
Financial backend	Mid	Low	High	Low
Moderation backend	Mid	Mid	Mid	Low
Subscriber DB	High	High	High	Mid

Caution: Avoid performing specific calculations to assess the threat to an asset. Even though we can define levels, assigning them some sort of specific numeric score is misleading and may mischaracterize the threats against a system.

Characterize How We Access Assets

- When do we read to a DB?
- Who is allowed to commit code?
- How does releasing code work?
- Where are our servers?
- What does normal trafic and use look like?

Limiting Anomalous Access

- Lock DB writes outside hours
- Reduce role access to create more “sign-offs” for code
- Ensure we have forward and backward models
- Limit geographical access
- Reject anomalous traffic

Trust vs Privilege

Privilege is how much you are able to do.

Trust is how much you are expected to be able to do.

Identify Trust and Trust Boundaries

1. Identify the privilege we wish to assign to an asset
2. Identify locations where communication between different privilege levels occurs
3. Document and plan for the ways in which privilege may be upgraded
4. Define these lines as trust boundaries
5. Assert a trust level betweeen boundaries

Model Potential Threats

Start with the first two of the four questions.

- What are we doing?
- What can go wrong?

Building on the previous section we covered, this should be something we're familiar with. If we need a higher degree of specificity because we're returning to an asset, this process can continue until we have a degree of granularity befitting the asset under question.

Attack Surfaces

As a rule, attack surfaces should be minimized. Performing this type of analysis is how we identify susceptible levels of an attack surface. The next step is to assess the risk and mitigate that risk as appropriate.

Threat Identification

After designating our attack surfaces, threat identification should be the straightforward next step. Anywhere the designated purpose of a system can be subverted can now be identified at appropriate levels.

Frameworks

STRIDE

What can be:

- (S)poofed
- (T)ampered with
- (R)epudiated
- (I)nformation disclosed
- (D)enied service
- (E)levated privilege

- Not a methodology
  - More apt: A *taxonomy*

Mapping Stride

Objective	Threat
Authenticity	Spoofing
Integrity	Tampering
Auditability	Repudiation
Confidentiality	Information Disclosure
Availability	Denial of Service
Authorization	Elevation of Privilege

More Examples

Threat modeling examples in the textbook:

- Bank Vault (pg. 33)
- Vegas Casino (pg. 37)
- sshd (pg. 31)
  - This example also covers the idea of trust boundaries

Risk

Types of Risk

- Financial
  - This will cost us money
- Disclosure
  - This will reveal confidential info
- Reputation
  - This will harm our reputation
- Legal
  - This will open us to legal liability
- Physical
  - This could impact physical assets
- Privacy
  - This could compromise privacy

Responses to Risk

- Assume the risk
- Transfer the risk
- Offload the risk

Transfer vs Offload

More coming in our section on mitigation

- Transfer
  - Insurance
  - Pay a provider
- Offload
  - Disclose
  - Delegate the risk to someone else

Aside: Presenting Risk

Our task will rarely be to make the final say on how to address some form of risk, as engineers, we're more likely going to need to lay out the risks and hand the decision over to the stakeholders. Presenting this risk is therefore a very important job of ours.

Outline These

- How the risk appears
- What has been done to mitigate it
- Why it can't be reduced further
- What would be needed to change the situation
- Who this impacts
- The potential outcomes
- A categorical likelihood of each outcome

Risk and Privacy

Taking on user data creates a risk of that data being disclosed. Kohnfelder says these fit into different types of consideration. We can also lump this under the security blanket of Confidentiality. Be aware, this is likely where any form of legal requirements are needed.

Mitigation and Prevention

Mitigation

Mitigation is the natural response to a world in which risk is guaranteed, but must be minimized. It is the methods by which we attempt to decrease the potential impact of a risk.

Threat Mitigation

1. Redesign or further defend to reduce the degree of harm presented by a threat
2. Remove the threatened asset
3. Transfer the risk to another group
4. Assume the risk

Redesign

This is the primary focus of our course, how to do this.

Additionally, this may be the only thing in your control as an engineer.

Remove

If an asset presents more risk than value, the decision may be made to discard the asset.

- Customer health data
- Tracking information
- Foreign servers

Transfer

- Insurance
- Service providers
- Outsourcing

Assume

Some threats don't need to be addressed. They may fall into the category of problems that we believe are okay to accept.

- Aliens
- Acts of God
- "Cost of doing business"

Goals of Mitigation

- Make harm less likely to occur
- Make harm less severe
- Make harm possible to undo
- Make it obvious harm has occured

Strategies for Mitigation

Minimize Attack Surfaces

This is the turtle. Harden the "outer shell."

- Move functionality "behind the walls"
- Minimize privilege escalation
- DevOps reduction

Migrate Functionality

If we consider every endpoint on a webserver as a "surface" which can be attacked by those who can view it, we can minimize (in other words mitigate their threat) by moving them out to the edge or put them behind some protection.

- Do extra calculation/processing on the edge
  - Perhaps query the raw data and do the processing there instead of having multiple different processing endpoints you could invoke
- Put queries behind a level of authentication
  - Require the standard `Bearer ...` token schema which is caught by middleware

Needed Escalation

If we consider an escalation as necessary in some instances, minimize the amount of work done by an escalated agent.

- Don't run `sudo` on every command
- Refer to `sshd` from Chapter 2.

DevOps Work

In the DevOps toolkit are a number of ways to reduce the potential attack surfaces.

- Firewalls and filtering
- SIEM (Security Information and Event Management)
- Isolation
- DMZs and Asset Segregation

Scoping

We're drawing the line here for the most part. Operations is its own problem which requires major investment and design just like the design of secure software requires major invesment and design. A few of the most prevalent Operations tools are mentioned because they can largely be assumed as present when designing software for running on the network, but as always there are tradeoffs and as always there are reasons to change plans or move the responsibitity from Ops to Design or vice-versa.

Narrow Windows of Vulnerability

Similar to minimizing attack surfaces, instead about minimizing attack time.

- Perform error checking/type checking before invoking privileged code to ensure as little time as possible is spent in a privileged environment
- See previous discussion on locking time windows

CAS

Code Access Security is a model which allows certain levels of privilege to be asserted and reverted from code before running specific aspects of code. Developed by the .NET v1.0 team (including Kohnfelder)

Minimize Data Exposure

Limit the lifetime of data in memory or even in storage.

Memory Vulnerabilities

Refer back to our discussion on Heartbleed for discussion on this class of issues.

Their existence is what has led to the idea of memory safe languages.

The path forward

Memory issues in software comprise a large portion of the exploitable vulnerabilities in existence. NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®. Memory safe languages provide

differing degrees of memory usage protections, so available code hardening defenses, such as compiler options, tool analysis, and operating system configurations, should be used for their protections as well. By using memory safe languages and available code hardening defenses, many memory vulnerabilities can be prevented, mitigated, or made very difficult for cyber actors to exploit.▪

Figure 1: The NSA announcement on using memory safe programming languages

User Deleting an Account Example

We run a system with user accounts which allow for the user to delete their account. Consider the following work flow:

1. User wants to delete their account
2. User deletes their account
3. Wait, actually I did that on accident

Mark for Deletion

1. User deletes their account
2. Account is not deleted for 30 days, instead just being marked for deletion
3. Email goes out to everyone with accounts announcing new feature
4. User is confused because they thought their account was gone

The Recommended Solution

1. User deletes their account
2. Push user account to separate, “deleted” database to be removed in 30 days
3. Delete the user from the standard database

Problems with this approach?

Applications of Mitigation

Access Policy

1. Start with a set of access needs
2. Design an ideal policy
3. Identify available options
4. Identify limits
5. Implement policy

Limits

Our application is a customer sales rep data collectino. There is information about who buys what and how much they spend for sales agents to sell further to the clients.

- One of our largest perceived threats is our customer data being leaked to a competitor
- Data being exfiltrated from a third party
- Data being exfiltrated by an insider (i.e. a sales rep)

Therefore, we want to limit access to the database to only a certain number of our customers per day

- 100 lookups max (i.e. new fiscal year)
- 20 lookups on average
- 10 lookups min

What is the number we should use for our access policy?

Problems with 100

If there are only a few days out of the year, this leaves us with a larger attack surface. If we ignored the days that we have a spike in usage, this would be overkill.

Problems with 20

The few max lookup days, which were probably already difficult days on the job, are now even more difficult because the policy gets in the way of people doing work.

Problems with 10

As can likely be assumed, this makes the every day job of people very difficult. This is a perfect example of sacrificing availability for confidentiality.

The Relief Valve

A possible means to mitigate the restrictions of this system are a way to temporarily elevate permission levels with a reason.

```
sudo apt update
```

Something which takes control over the machine, elevating permissions, for a specific purpose provided some small level of authorization.

Alternatively, a system which allows for restrictions to be set based on some other metric. For instance, if we see way more calls coming in, continue increasing the limit until the influx of callers subsides.

Our solution which allows for elevation with reasoning allows very easy auditability. Further, we can update the policy if we see more and more overrides coming through that we judge to be legitimate.

Interfaces

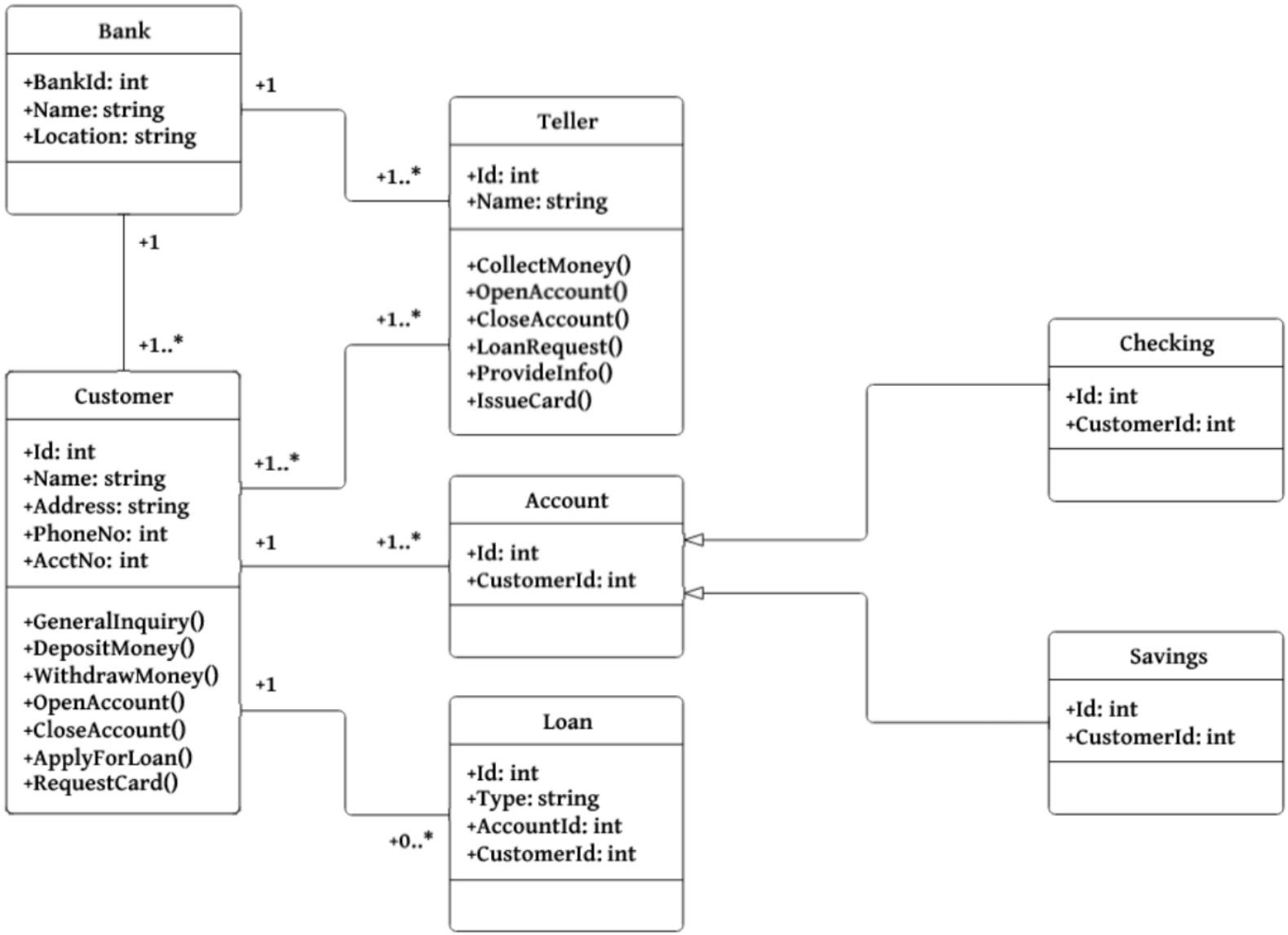

Figure 2: A software layout presented via UML

Trust Boundary and Information Flow

Each interface is the “bottleneck” of information flow. It’s very easy to determine which interfaces carry different levels of privileged information and focus responses on them.

Three different interfaces:

- Login interface which needs to authenticate with username and password
- Profile update interface which carries modifications to user settings/data
- Webpage `get` interface

From these three interfaces, it's easy to understand which should have attack surfaces mitigated first.

- Login
  - Ensure encrypted transmission
  - Temporarily store, salt, and hash the password
  - Immediately discard
    - Clean up memory in a safe way

Network Interfaces

- gRPC
- HTTP/S
- GraphQL
- REST/CRUD

1. Use encrypted transfer over HTTPS
2. Rely on a CA for signing/trust
3. Don't reinvent the wheel

Communication

- Air-gapped devices are virtually non-existent
- Everything has the network as a potential attack surface
  - and potentially also the underlying device
- The wire is also an attack vector
- Fact of communication always visible

Storage

Storage of sensitive data is “much like communications because storing data is like communicating it with the future.”

- Storage medium is the “wire”
- Larger window for vulnerability

Backups

Different data needs backed up in different ways. When should we back up:

- our local gitlab instance?
- the slack server?
- our encryption and signing keys?
- our website log files?
- our payment processing log files?
- our user database?

3-2-1 Backup Rule

- There should be 3 copies of the data
- It should be on 2 different media
- At least 1 copy should be offsite

Returning to the sales rep application example.

1. Production DB
2. Onsite backup
3. Deep storage

Storage is Dirt Cheap

S3 Glacier Flexible Retrieval (Formerly S3 Glacier)***- For long-term backups and archives with retrieval option from 1 minute to 12 hours	
All Storage / Month	$0.0036 per GB
S3 Glacier Deep Archive*** - For long-term data archiving that is accessed once or twice in a year and can be restored within 12 hours	
All Storage / Month	$0.00099 per GB

Figure 3: AWS S3 Glacial Storage

Physical Media

On the subject of backups, the physical media is a real, meaningful problem that must be addressed.

- degradation
- availability
- obsolecense

Storage Tradeoff

"[Data storage] is a fundamental tradeoff that requires you to weigh the risks of data loss against the risk of leaks." - Kohnfelder (pg. 51)

Final Takeaways About Threats

1. Identify attack surfaces
2. Model threats on the system
3. Assess risk posed by attacks
4. Mitigate and prevent where possible

Questions?

Next Time

- Attacking Systems
- Finding Vulnerabilities in Code
- Finding Vulnerabilities in Design