# *Architecting a secure system*

*6.1600 Course Staff*

*Fall 2023*

So far, we have been focusing on security for network communication. We have established many tools to achieve this, from message authentication codes to public-key encryption.

Ultimately, however, applications need to make use of these tools. And for our network security tools to provide meaningful security, the applications themselves must be reasonably secure.

In discussing platform and application security, there are two classes of problem that we want to defend against.

**Disclaimer:** This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

1. *Mistakes of various types.*

- Buggy systems: including hardware and software bugs
- User mistakes: phishing, misconfiguration

2. *Malicious people or components.*

- Malicious components: malware, supply-chain attacks
- Malicious users: what if the adversary gets the admin's password?
- Attacker gets access to the system: insider attacks, the adversary guesses credentials,

In computer security, we tend to treat mistakes/bugs and malicious software/components in the same way. We do that because (1) it's often difficult to specify what it means for a component to be "non-maliciously buggy" and (2) an attacker can often leverage what seems like a benign bug into full-fledged misbehavior.

Thus, a theme that will be present throughout this section is that we will consider mistakes to be malicious: if we are prepared to handle malicious components, we will similarly be prepared to handle our own buggy code. If we are prepared to limit damage of a malicious user with the admin password, we will also be limiting the damage that a mistake-making admin can cause.

This multitude of threats makes designing secure applications quite difficult. To make progress, we will seek to design systems that limit damage when things go wrong.

When we design for security, we have essentially three goals:

1. **Defend against known attacks.**
2. **Defend against unknown attacks.**

3. **Limiting damage.** In the cryptography part of this course, systems are either security or insecure. In systems security, things are much more gray. Attacks we care about often are outside of our threat model—even when this happens, we'd like to somehow contain the damage.

## 1 Isolation

One of the most effective strategies to limit damage is to split a system into isolated components. If one of these components becomes compromised, it should not be able to compromise the other components. For example, if you run code in one virtual machine, it should not be able to tamper with data in another virtual machine.

When choosing what mechanism to use to isolate various components, we think a lot about the *performance overhead* of an isolation mechanism. The challenge of building a good isolation mechanism is ensuring strong isolation without slowing down the isolation components too much (or taking up too much extra memory).

These components will typically run on top of some *host* that enforces isolation. Importantly, this host must be correct! If there are bugs in the host, malicious code in a component may be able to exploit a bug to escape its isolation. The success of an isolation mechanism depends on the correctness and configuration of the host.

| Examples | Host |
|---:|---|
| Docker Container | Operation System (e.g. Linux) |
| Browser tabs | Browser |
| Language-Level (JavaScript, Wasm) | Language Runtime |
| Process | Linux kernel |
| Virtual machines (VMs) | VM Monitor |
| Physical ("air gap") | Physics |

Table 1: Some common types of isolation

In order for these isolated components to be useful, they will need to be able to talk to each other in some form. For example, a client component must be able to make requests to a database component, but we would like to limit the power of the client to do damage. For this, we would like to achieve *controlled sharing*.

### 1.1 Controlled Sharing

For an isolation mechanism to be useful, it additionally needs to have some way to interact with other isolated components. For example, some JavaScript code isolated in a browser tab still needs some means by which to make requests over the network.

When a host decide whether to allow a request from a particular component, it typically needs to do three things with each request:

Since all three of these actions start with the letters "Au," we sometimes call this the *gold standard* for controlled sharing.

- Authenticate: Associate the request with some *principal*. A principal could be a user name, an "origin" in the web context (e.g., `google.com`), a program, or some other entity in the system.

- Authorize: Decide whether that principal is allowed to make the request.
- Audit: Keep track of requests that each principal makes. Auditing is about limiting damage: often a host will mistakenly allow requests it shouldn't; audit logs make it easier to discover such mistakes and to clean up afterwards.

It is crucial that an isolation mechanism perform these three checks on every single request—a single hold in the isolation boundary is often enough to completely break any benefits isolation that would have provided.

## 2 *Authentication*

Since we already had an entire module on authentication using signatures, MACs, passwords, and so on, we will not discuss authentication further here.

## 3 *Authorization Policies*

In order to authorize requests, we need some sense of permissions—a mapping from *objects* to *principals* that can access them. We call these permissions the authorization *policy*.

*Storing policies.* We can think of an authorization policy as a gigantic matrix with one row per object and one column per principal. For example, in a file system, we could have one row per file, and one column per user in the system. The entry in column $i$ and row $j$ lists the actions that user $i$ can perform on file $j$: read, write, execute, etc. A common way of storing this gigantic matrix, for example in AFS, is via an *access control list* for each object.

*Setting policies.* There are many approaches to setting authorization policies. As always in computer systems, there is no one perfect solution:

- **Discretionary access control: "Owner" of each object sets the policy.** This approach is useful in file systems—each file has an owner and the owner can determine who has access to the file. A problem is that if an attacker hijacks the owner's account (or just one application that the owner runs), the attacker can tamper with the policy for all of the user's files. In addition, it may be difficult for non-expert users to set policies.

- **Mandatory access control: Administrator sets policy.** This approach often is useful in a large organization, when administrators have opinions about which user should have access to which files or systems. A classic example of this is for systems that handle classified data in government systems. Normal users of the system cannot give unprivileged users access to a classified files.

  One limitation of this approach is that it is very coarse grained: administrators may not know exactly who should have access to what.

Systems in practice often use some combination of both of these strategies.

*Role-based access control* tries to hit some midpoint between discretionary and mandatory access control. In these systems, there is a centrally defined set of "roles." In a university, these could be "Students," "Faculty," and "Staff." The security administrator assigns users to roles. Then application developers determine which roles have access to the application.

Common issues are:

- It is difficult to keep policies up to date as the set of users evolves. Expiring permissions is one strategy.
- Users will complain if they do not have enough permissions, but they will never complain if they have too many permissions. As a result, users often end up with more access than they need from a security standpoint.

## 4 *Auditing*

We have relatively little to say about this. The most important thing to remember about auditing is that a system should store the audit logs in a container that is separate from the container holding application logic. That is important because if the attacker compromises the application, it should be difficult for the attacker to compromise the logs as well.

## 5 *Delegation and Chained Requests*

Users often interact with systems *indirectly*. For example, when accessing Gmail, a user's browser first sends a request to the Gmail server asking for new messages. The Gmail server then sends a request to the database to fetch the message data.

For the first request, it is fairly clear that the principal should be Alice: the request is coming from Alice's browser, and therefore should have been initiated by Alice directly. Alice will send some credential to the server, and the server can use this credential to verify that it is really Alice on the other end. For the second request, however, it is not as clear who the request should be from.

One option is to have the request come from Alice. This protects against compromise of the Gmail server—the adversary cannot see

all user data. Systems like SSH and AFS follow a strategy like this. Another option is for the principal of this second request to be the Gmail server itself. This helps with isolation among services access the same database: if the Google calendar code is buggy and gets compromised, the first plan would allow an adversary to view Alice's gmail data even is the gmail service was perfectly secure. However, it does not protect other users from a buggy Gmail service.

*Compound Principal: "B for A."* To achieve something stronger, we can create a new type of *compound principal* that combines a service or device with a user. For example, this server-to-database request could carry a principal of "Gmail Server for Alice". This provides protection against both gmail server compromise and against compromise of other services.

However, it is not as clear how to actually implement this. One option is to continue to have Alice send her credential to the server directly. However, then the server can totally impersonate Alice and we gain little protection. What we would really like is for Alice to give permission to the Gmail server to fetch her emails, but not to do anything else. This is called *delegation*.

*Delegation with cryptography.* In interacting with the Gmail server $B$, we may like for Alice ($A$) to give $B$ permission to authenticate as "$B$ for $A$" and to do so for only 60 seconds into the future. To achieve this, $A$ can sign a message that outlines the permission it would like to give to $B$. This signature becomes the proof of authorization. As an example:

$$\mathsf{Sign}(\mathsf{sk} - A, \text{“A delegates to B”}, \mathsf{start} = \mathsf{now}, \mathsf{end} = \mathsf{now} + 60)$$

Google indeed uses a strategy like this. They have a global DoS-resilient HTTP front-end that performs initial authentication. This frontend is then responsible for generating these scoped delegation signatures for each operation that the user would like to do and sending them along to the individual services. These signatures are then used for all following operations.

*Capabilities.* We may want more fine-grained access control. For example, on Android, the Gmail app may like to delegate permission to a PDF viewer to view an attachment. However, if all the attachments are stored in some common database, we would like to avoid giving the PDF viewer access to view everything in the database. To achieve this, Android (and systems more generally) use a plan called cababilities.

A similar strategy can be seen, for example, in cloud file sharing: when you share a file in google drive, it generates a long random link that allows anyone with access to that link to view that file (and no others). This link itself becomes a *capability*—it allows anyone that posesses it to perform some related action.