The Clipper Chip

For much of the 20th century, cryptography was heavily controlled by the US government (its teaching, practice, etc.).

The **Clipper Chip** was a piece of software designed by the government to provide **key escrow**. In particular, the system was a chip that used strong symmetric encryption. The key idea was that law enforcement entities could decrypt messages sent through the Clipper Chip. The choice was to either not use strong cryptography (because it was illegal), or use strong cryptography but allow the government to decrypt your messages.

The Law Enforcement Access Field (LEAF) was a key part of thee design, with the Clipper Chip functioning as follows:

1. Sender and receiver negotiate a key k_s (e.g., with Diffie-Hellman).
2. Sender sends $E(k_s, \text{message}), LEAF$ to the receiver.
3. The receiver checks the LEAF.
4. The receiver decrypts.

In more detail, the sender would come up with a session key. The Clipper Chip would then provide a unique unit key, and these would go into an encryption scheme. This would be concatenated with a unit ID and a checksum, computed as the hash of the session key and some parameters. The output would be 128 bits, which would then be encrypted again with a family key (common to all chips). The output of this encryption is known as the *LEAF*.

The receiver then receives the LEAF, decrypts with the family key, checks that the checksum works out, and proceeds to decrypt.

This scheme was broken. Notice the following:

- The checksum was only 16 bits, so with 2^{16} tries you could brute force it, which is a searchable space.
- The checking algorithm is the very same on both the sending and receiving end. Therefore, you could send the same message to your *own* Clipper Chip and check if it works. As a result, you would only have to send one message to the target chip because you can verify the checksum beforehand.

Secure System Design

Systems are only as secure as their components. With secure systems, you want your components to be isolated with nuanced access control.

Access control

Access control is a combination of authentication (i.e., who is asking?) and authorization (i.e., do they have authority?).

We've discussed authentication, so now we'll look at authorization. There are typically a few approaches here:

- Access control matrix or list (e.g., a list of who has access to what).
- Capabilities.

Access control matrix

The basic operation is that *subject* wants to do *verb* on *object*. Do we allow it?

The policy is the set of allowed (s, v, o) triples. But how is the policy set? And how is it *enforced*?

The subject is often a process and the object is often some resources, file, open network connection, window, etc. We ten to use labels to simplify a policy, e.g., label a process with a user ID and set the policy based on labels.

Labeling can be complicated. Suppose Alice runs a program written by Bob. How do we label them? If we treat it as Alice, then the program can steal Alice's data. If we treat it as Bob, then Alice can read Bob's files.

A common OS approach (e.g., in Linux) is to use a *setuid* bit, so Bob decides whether the program runs as himself or the invoker.

Storing the policy information

We use a **access control matrix** (ACL), where the columns represent objects, the rows represent subjects, and $M[i][j]$ is a list of allowed verbs.

But who sets up the ACL? A centralized, top-down policy (e.g., the registrar decides who can see whose grades) has the advantage of being done by a well-trained person. But it's very inflexible and slow. A decentralize approach would be to give each object an owner and have to owner set the ACL. The advantage here is that it's very flexible, but it's also mistake-prone.

Groups

Logically, we can divide users into sets, i.e., groups. We can then give access to groups, rather than to individuals. The advantage is that the use of groups makes ACLs shorter and easier to understand. It also encodes the *reason for access*.

Roles

If a person "wears several hats", have a role for each "hat"; allow users to step in and out of their roles. For example, Professor Felten could have a CITP hat and a separate Wilson School hat.

Traditional Unix file access

Traditionally, a file in a Unix system belongs to one user or one group. The ACL for each operation contains a subset of {user, group, everyone}. If the *setuserid* bit is set to true, then it executes as the file owner; otherwise, it executes as the invoker.

Capabilities

The core operation is that "the bearer may do *verb* on *object*." A different way to look at permissions.

The implementation is usually cryptographic, i.e., you store (verb, object, $PRF(k, verb||object)$). Alternatively, the OS can keep track of things.