15-316: Software Foundations of Security and Privacy **Spring 2017**

Lecture 16: Audit

Lecturer: Jean Yang

We will begin with a couple of group exercises. Consider the following systems:

- **Social network.** Think of something like Facebook, where thousands of software engineers are writing code that is running on machines all over the world, users are sharing their personal data, and there is incentive to steal this data in all kinds of ways.
- **Health record manager.** Like a social network, but with more regulatory incentive to protect data.
- **Voting system.** Imagine a county election system where there are voting machines, voting machine operators, ballot count inspection people, etc. [We will work together in class to figure out what the model should be.]

Let us now discuss the following questions:

- What are the policies we want to apply? (What are the policies we are *required* to apply by law?)
- What are hacks we want to prevent against?
- Where do the techniques we've learned so far help?
- What do we have to *trust* in order to apply these techniques? When can we trust these entities, and when can't we?
- What parts of the policies can be formally specified? What parts cannot, and how do we handle those?

For the discussion, we will keep in mind the techniques we have learned so far this semester:

- Language abstractions for making systems safer.
- Testing and formal verification.
- Safety properties and reference monitors.
- Authorization logics.
- Systems-based and language-based information flow techniques, and declassification.

[In-class discussion about where the gaps are.]

16.1 Introduction to Audit

In the above discussion, we should have concluded that there are elements of the systems that we cannot formally model. There are also elements of the system that we may not have accounted for in our model, and we may need to piece together what happened after the fact.

It turns out that there are three classes for accountability mechanisms:

- **Authorization.** Mechanisms that govern whether actions are permitted, for instance file permissions and visibility restrictions on Facebook.
- **Authentication.** Mechanisms that bind principals to actions, for instance passwords and digital signature.
- **Audit.** Mechanisms that review actions and who took them, for instance log files and intrusion detection systems.

So far, we have learned about authorization and authentication. Today, we will discuss audit. In computer security, audit has the following purposes:

- **Individual accountability.** Logged actions can be reviewed to hold users accountable for their behavior.
- **Event reconstruction.** After an attack is detected, audit can establish what occurred, and what the damage was.
- **Problem monitoring.** Real-time audit can help monitor problems as they occur.

Typically, people mean something fairly specific when they talk about audit, but let us think about audit from a first-principles point of view:

- What are the goals of audit? (What can audit achieve? Who are we protecting with audit?)
- What is fair game for audit?
- What kinds of techniques may be useful for performing audit?

For the rest of this class, we will focus on auditing logs from program execution, rather than auditing code.

16.2 Logging Access

To demonstrate what logs can be useful for, consider this story a hacker friend Sz told me about a time he was asked to investigate what looked like a hack. The client was a service that regularly surveyed customers on its mailing list, and there had recently been a mass unsubscription of users.

The setup is as follows. Their systems were a "hodgepodge of logged, monitored, or not," and they had previously found a vulnerability in their system. The client gave Sz "a few gigs of log data," and Sz proceeded to try to model what was going on mentally. None of the data was complete: firewall logs didn't have protocol information, server logs didn't have external IP information, and proxy logs were incomplete.

Sz proceeded as follows:

- Sz begins to dig through the proxy logs, and notices IP addresses for Central Asia, Russia, etc. and discovers that there was a bot net hung up on one of the pages, but not having much success. So this wasn't the problem.
- Sz finds users issuing requests with ROT13 "encrypted" strings, representing clear SQL injection attempts. But this wasn't the problem either.
- Sz finds users registering and unregistering hundreds of times per month, but without clear reason.

- Sz goes back to thinking about how the site was lays out. The logs provided the rough time frame for the lack, so Sz started doing frequency analysis of the number of times pages were called, and what HTTP method was used each time. Sz noticed the unsubscribe page was being hit frequently... by certain mail systems and anti-viruses, with a `HEAD` method. This led Sz to think about the application work flow: a user subscribes to a mailing list and can unsubscribe at any time. Sz began to wonder why unsubscribe was being hit so frequently, and why it was a `HEAD` Method, and realized: many email systems and anti-viruses will poll resources sent in emails to check if they are "alive" and contain viruses. In this case, polling the unsubscribe endpoint involved unsubscribing the user. Sz's client only realized this when there was a mass unsubscribe event, which in this case coincided with one of their large user installations switching anti-resources. According to Sz: "They had had the problem for years, but never realized it, because it never impacted a large number of users."

In finding this bug, Sz ended up finding six different attacks against the application, a large amount of abuse, and one terrible design flaw. And this could not have been done without audit logs!

Here are some discussion questions:

- What was the problem? What could have prevented this problem in the first place?
- What kinds of audit information did Sz have access to? What was the purpose of audit in this case?
- What was Sz trying to reconstruct from the audit logs? What information would have made it easier to reconstruct this information?
- How does this story illustrate security challenges people face in the real world?
- What did Sz actually get from the logs in the end? How much of this could be automated? What would be difficult to automate?

More generally, there are two main tasks for this style of audit:

- **Logging.** Recording events or summary information (for instance, statistics) corresponding to system use and performance. The challenge is to record enough information to infer (attempted) violations of security policies.
- **Reviewing.** Analyzing log records to detect (attempted) violations of security policies.

16.2.1 What to Log

In deciding what to log, system designers need to strike a balance between recording too much information and too little information. In designing systems, we need to reason about both the *functional requirements* and the *security requirements*. Both requirements stipulated *testable conditions* that suffice for an action to be permitted, and can be either verified before the system runs, or audited after the fact.

Logging can capture system states, or individual events:

- **States.** This involves taking a snapshot of key data, and allows recovery when a system fails. It can be difficult to take a consistent snapshot.
- **Events.** Logging important actions is more common for security purposes. For instance, the Windows security log records events corresponding to account login, access to operating system resources, change to security policies, and exercise of heightened user privileges.

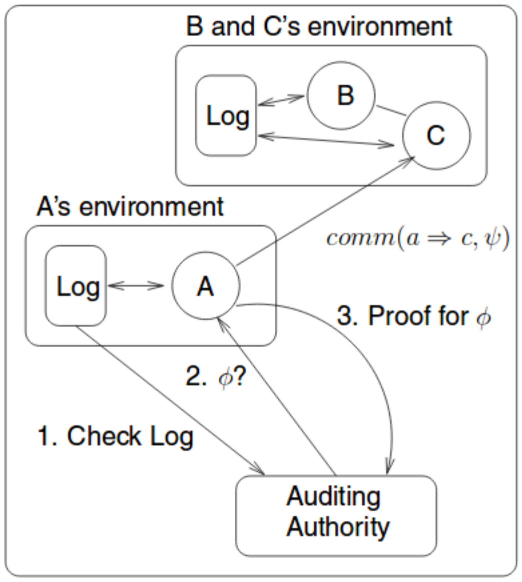

Figure 16.1: Framework for audit logic.

The Orange Book stipulates the following minimal (level C2) logging requirements:

> The TCB shall be able to record the following types of events: use of identification and authentication mechanisms, introduction of objects into a user's address space (e.g., file open, program initiation), deletion of objects, and actions taken by computer operators and system administrators and/or system security officers, and other security relevant events. For each recorded event, the audit record shall identify: date and time of the event, user, type of event, and success or failure of the event. For identification/authentication events the origin of request (e.g., terminal ID) shall be included in the audit record. For events that introduce an object into a user's address space and for object deletion events the audit record shall include the name of the object. The [system] administrator shall be able to selectively audit the actions of any one or more users based on individual identity.

16.3 Logic for Audit

It turns out that some of the mechanisms for checking a program *before* they run can also help with checking logs collected while a program is running. In class today, we will cover an audit logic for accountability that looks very much like the proof-carrying authentication we saw earlier this semester. The rest of this lecture is based on the paper "An Audit Logic for Accountability" [1]. Their framework is based on the model in Figure 16.1.

16.3.1 Basic

The system has the following components:

- *Agents* $a, b, c \in \mathcal{G}$.

- *Agent variables* $A, B, C \in \mathcal{V}_a$.
- *Data objects* $d \in \mathcal{D}$.
- *Permissions* and *facts* $p \in \mathcal{C}$. Examples include $\mathsf{read}(a, d)$ and $\mathsf{partner}(a, b)$.
- *Actions* $act \in ACT$. We assume two actions are always present: communication of policies $\mathsf{comm}(a \Rightarrow b, \phi)$ and data creation $\mathsf{creates}(a, d)$.

16.3.2 Policy Language

Policies $\phi \in \Phi$ are defined as follows, where the owns relationship indicates the owner of an object, the says relationship expresses when one agent is allowed to give a policy to another:

$$\begin{array}{lll} \phi & ::= & p(s_1, \ldots, s_n) \\ & & \mid A \ \mathsf{owns}\ D \ \mid A \ \mathsf{says}\ \phi \ \mathsf{to}\ B \\ & & \mid \phi \wedge \phi \ \mid \forall x.\phi \ \mid \phi \rightarrow \phi \ \mid \xi \rightarrow \phi \\ s & ::= & A \ \mid D \\ \xi & ::= & !act \ \mid ?act \end{array}$$

The notation with actions designate whether they need to happen each time, or only once. Example policies are as follows:

1. The policy that allows Bob to read every data object owned by Alice is $\forall x.(a \ \mathsf{owns}\ x \rightarrow \mathsf{read}(b, x))$.
2. Let $\mathsf{age21}(x)$ denote that agent x is at least 21 years old, and $\mathsf{alc}(y)$ denotes that beverage y is alcoholic. A policy allowing people over 21 to drink alcoholic beverages is $\forall x, y.(\mathsf{age21}(x) \wedge \mathsf{alc}(y)) \rightarrow \mathsf{drink}(x, y)$.
3. If we require a payment \$10 on the previous permission, the policy becomes $\forall x.(!\mathsf{paid}(x, \$10) \rightarrow \forall y.(\mathsf{age21}(x) \wedge \mathsf{alc}(y)) \rightarrow \mathsf{drink}(x, y))$.

16.3.3 Actions and Permissions

We label each instance of an action with a unique identifier id, and this formally gives a set $AC = \mathbb{N} \rightarrow ACT$ of "executed actions" or "action instantiations." The following functions describe three relevant properties of actions:

- The *observability* function $obs : AC \rightarrow P(\mathcal{G})$ describes which agents can observe which actions.
- The *proof obligation* function $po : (AC \times \mathcal{G}) \rightarrow \Phi \cup \{\bot\}$ describes which policy an agent needs to justify the execution of an action, where $\bot$ describes the null policy.
- The *conclusion derivation* function $concl : (ACT \times \mathcal{G}) \rightarrow \Phi \cup \{\bot\}$ describes what policy an agent can deduce observing an action.

While the observability and proof obligation functions depend on executed actions, the conclusion derivation function is purely syntactical.

For the default actions $\mathsf{creates}(a, d)$ and $\mathsf{comm}(a \Rightarrow b, \phi)$, we have:

$$
\begin{array}{rcl}
obs(\mathsf{creates}(a,d)) & = & a \\
obs(\mathsf{comm}(a \Rightarrow b, \phi)) & = & \{a, b\} \\
\hline
po(\mathsf{creates}(a,d), a) & = & \perp \\
po(\mathsf{comm}(a \Rightarrow b, \phi), c) & = & \perp (a \neq c) \\
po(\mathsf{comm}(a \Rightarrow b, \phi), a) & = & a \textsf{ says } \phi \textsf{ to } b \\
\hline
concl(\mathsf{creates}(a,d), a) & = & a \textsf{ owns } d \\
concl(\mathsf{comm}(a \Rightarrow b, \phi), a) & = & a \textsf{ says } \phi \textsf{ to } b \\
concl(\mathsf{creates}(a,d), b) & = & \perp (b \neq a) \\
concl(\mathsf{comm}(a \Rightarrow b, \phi), c) & = & \perp (c \neq b)
\end{array}
$$

16.3.4 Proof System

We have the following proof system that contains the standard predicate logic rules, and the following additional rules:

$$
\textsc{Say}\ \frac{b \textsf{ says } \phi \textsf{ to } a}{\phi}
\qquad
\textsc{Refine}\ \frac{\phi \rightarrow \psi \quad a \textsf{ says } \phi \textsf{ to } b}{a \textsf{ says } \psi \textsf{ to } b}
\qquad
\textsc{Obs\_act}\ \frac{act \quad concl(act, a) \neq \perp}{concl(act, a)}
\qquad
\textsc{Der\_pol}\ \frac{a \textsf{ owns } d_1 \ldots a \textsf{ owns } d_n}{\phi[\{d_1, \ldots, d_n\}]}
$$

Example. Suppose we have a predicate $\mathsf{rel}(d, \overline{d})$ expressing whether two data objects are related, for instance a review of a new product and a press release. Alice creates d and wants to give a policy $\forall x.\mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)$ to Bob, giving Bob to print a document as soon as a related object exists. Alice can build the policy as follows:

$$
\textsc{Refine}\ \frac{
\textsc{Implies-I}\ \dfrac{
\textsc{Forall-I}\ \dfrac{
\textsc{Implies-I}\ \dfrac{[\mathsf{print}(b,d)] \quad [\mathsf{rel}(d,x)] \quad \mathsf{print}(b,d)}{\mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)}
}{\forall x.\mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)}
}{\mathsf{print}(b,d) \rightarrow \forall x.\mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)}
\quad
\textsc{Der\_Pol}\ \dfrac{
\textsc{Obs\_Act}\ \dfrac{\mathsf{creates}(a,d) \quad concl(\mathsf{creates}(a,d), a)) = a \textsf{ owns } d}{a \textsf{ owns } d}
}{a \textsf{ says } \mathsf{print}(b,d) \textsf{ to } b}
}{a \textsf{ says } \forall x.\mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d) \textsf{ to } b}
$$

16.3.5 Modeling the System

A logged action is a triple $lac = \langle act, conds, obligs \rangle$ consisting of an action $act \in AC$, a set of atomic predicates $conds$, and a set of labelled annotated actions $obligs \subset \{!, ?\}AC$. When logging an action, an agent can include supporting conditions that the environment certifies to be valid at the time of the action in the set of predicates $conds$. The obligations in $obligs$ refers to other actions the agent did or promises to do. We assume there is a way of checking if actions have expired. For example, if the agent promises to pay within a day, then a payment action needs to be done and logged within a day of logging the action.

Let's continue with our example. Suppose we introduce an action $\mathsf{drunk}(x,y)$ and a corresponding atomic predicate $\mathsf{drink}(x,y)$, with $concl(\mathsf{drunk}(x,y), x) = \perp$ and $po(\mathsf{drunk}(x,y), x) = \mathsf{drink}(x,y)$. We also introduce an action $\mathsf{paid}(x,y)$ with corresponding atomic predicate $\mathsf{pay}(x,y)$ with $concl(\mathsf{paid}(x,y), x) = po(\mathsf{paid}(x,y), x) = \perp$. We can express the following logged action for payment:

$$lac_{\mathsf{pay}} = \langle \mathsf{paid}_0(a, \$10), \emptyset, \emptyset \rangle$$

Here is the action for drinking a beer:

$$\begin{aligned} lac_{\mathsf{drunk}} = &\langle \mathsf{drunk}_1(a, beer), \\ &\{\mathsf{age21}(a), \mathsf{alc}(beer)\}, \\ &\{!\mathsf{paid}_0(a, \$10)\}\end{aligned}$$

The log of an agent a is a finite sequence of logged actions, with the following consistency properties:

- An agent logs an action at most once.
- An agent can include the same obligation at most once within the obligations of logged actions.
- An agent cannot log an expired action.

A system is a six-tuple:

$$\langle \mathcal{G}, \Phi, ACT, obs, concl, po\rangle$$

A state is $\mathcal{S}$ is the collection of logs of the different agents. Agents who observe agents may choose to log actions. An execution of a system consists of a sequences of transitions $\mathcal{S}_0 \overset{act_1}{\rightarrow} \mathcal{S}_1 \overset{act_2}{\rightarrow} \cdots \overset{act_n}{\rightarrow} \mathcal{S}_n$, and the execution trace is $act_1\ act_2 \ldots act_n$, projecting only the logged actions.

Agents may be audited by some *audit authority*, at state $\mathcal{S}$. Given $\mathcal{S}(a)$ and the observed actions S, the authority should be able to orderly properly the actions and check *justification proofs*. Please see the paper for more details!

16.3.6 Discussion

Discussion questions:

- How does this kind of work address the problems that Sz talked about?
- How do we need to complement this kind of work to make it practical?

References

[1] R. Corin, and S. Etalle, and M. A. C. Dekker, and J. I. den Hartog, and J. G. Cederquist. An audit logic for accountability. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 00:34–43, 2005.