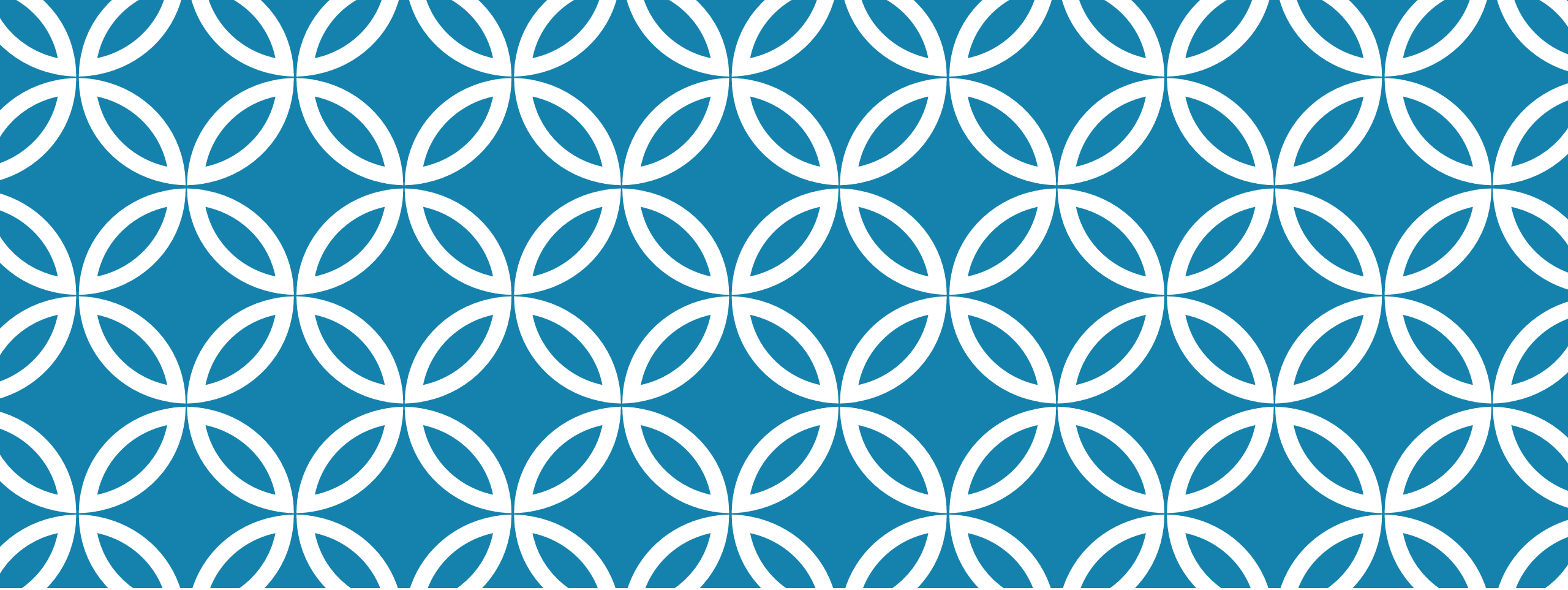


ACCESS CONTROL ABSTRACTIONS

Professor Ken Birman

CS4414 Lecture 26

IDEA MAP FOR TODAY

Authentication, Authorization, Access Control

Access Control Lists. Capability models.

Security Enclave Model

Information Flow Monitoring and Control

OUR TOPIC TODAY, IN "PLAIN WORDS"

Think of a doctor's office: patient data must be protected, yet also needs to be shared with authorized individuals. This pattern arises in many applications.

In order to secure data or protect privacy, we need to be able to express what we are trying to accomplish.

What tools exist for doing this?

HISTORY OF ACCESS CONTROL CONCEPTS

Early systems simply had a user id and password.

Then Linux introduced groups, for people collaborating in teams

But modern systems have needed steadily more structure and protection, leading to increasingly elaborate options

ACCESS CONTROL: WHO, WHAT, WHEN, HOW?

At the core of access control is a basic question: We have some actor, and some resource. The actor wishes to perform an action on the resource. And we ask:

- Who controls this actor?
- What resource is this, and who owns it?
- What policy was specified for access?
- Does that policy permit this access at this time?

USE OF ACCESS CONTROL MODELS

In the actual systems, these models "inspire" the interfaces and are the proper way to think about our goals and policies.

Theoretical work on security often starts by formalizing access control in a mathematical notation, at which point one can prove theorems about policies, and even build verifiers to confirm that a program is compliant with the theory (like type checking!)

THE SYSTEMS VERSION OF THIS QUESTION ARISES AT MANY LEVELS

In Linux, access control is applied when a process wishes to read or write data from a source (including the kernel, or /dev/proc, or a device)

In C++ it arises whenever code accesses an object in some way.

Networking extends the question to a whole distributed system!

AUTHENTICATION: WHO'S WHO?

Abbot and Costello: "Who's on first?"

The procedure that the kernel uses to decide who this process belongs to is called *authentication*.

One user can own many resources and processes. A new process normally inherits the same user id and group id as its parent, but in fact the parent can remove the group id and in some situations can even remove or change the user id.

Example: When you log in, /etc/init (running as root) launches a bash shell running with your user and group ids (using setuid, setgid).

TO SHOW WHO YOU ARE, A PASSWORD IS OFTEN NOT ENOUGH

We all know how easy passwords are to steal or guess

This has led towards methods that are much stronger and either don't require a password or, more often, add extra things beyond what the password provides.

CREDENTIALS, TWO-FACTOR AUTHENTICATION

Linux centers on the idea of a user-id that is validated either when you log in and type in a password, or perhaps with some secondary mechanism (fingerprint, RSA code, text-message code)

There are also cryptographic key-based credentials. These often split into a public key for talking "with" you and a private key that only you control.

CREDENTIAL STORAGE: CERTIFICATE REPOSITORIES

Most systems have some form of secure credential storage service.

When you create a key (for example, withssh-keygen) it generates a file with the public and private portions.

You store the public part into a *certificate* and upload it to a repository. Now anyone can download this public data.

PUBLIC AND PRIVATE KEYS

Two functions… one is the inverse of the other.

$K_{private}(X)$: X is encrypted using your secret private key.

$K_{public}(X)$: X is encrypted with your public key

$K_{private}K_{public}(X) = X$: private decrypts public.

$K_{public}K_{private}(X) = X$: public decrypts private

HOW THESE CAN WORK IN A CHALLENGE SETTING

Ken: I wish to log into system A

A: $K_{ken\text{-}public}$("To prove that you are Ken, send me 1718981")

…. *only Ken will be able to decrypt and read this*

Ken: $K_{ken\text{-}private}$("Here's my proof: 1718981")

… *only Ken would be able to send this proof*

SPOOFING

But… what if someone tries to spoof by pretending to be A?

Then Ken could be tricked into thinking he was talking to A.

We can solve that with one more step…

HOW THESE CAN WORK IN A CHALLENGE SETTING

Ken: I wish to log into system A

A: $K_{A\text{-private}}$ $K_{ken\text{-public}}$("To prove that you are Ken, send me 1718981")

…. only Ken can read this. Only A could have sent it.

Ken: $K_{ken\text{-private}}K_{A\text{-public}}$ ("Here's my proof: 1718981")

… only A can read this. Only Ken could have sent it.

HOW THESE HELP WITH AUTHORIZATION

Tammy: Ken, please get some printer paper from Gates 302

$K_{Tammy\text{-}private}$("I authorize [ken] to [enter] [Gates 302] [during the next five minutes] to [get] [printer paper]")

… The statement in the certificate describes what I am authorized by Tammy to do. The signature "proves" that Tammy issued it.

SIGNATURES VERSUS ENCRYPTION

Notice that we didn't encrypt the certificate. It is slightly costly to encrypt large objects, but more to the point, encryption hides even fields like "To" and "From".

So some systems do encrypt entire objects.

Signatures first compute a hash of the object, then encrypt the hash. Tampering will break the pairing.

THUS, DIGITAL SIGNATURES ARE A TOOL

In effect, with a notion of entities, we can assign unique public-private key pairs to each entity.

Then we can create certificates signed by the entity that make statements that could be verified digitally and used as the basis of an authorization scheme.

As a user, you would authenticate yourself and the entity could then issue you a certificate saying “the holder of this logged in as Ken”

SOMETHING YOU KNOW… SOMETHING ABOUT YOU… SOMETHING YOU HAVE…

You **know** your password, or perhaps have a file with your private key in it.

An image of you, or a fingerprint, or your DNA: **things true about you** that a system could validate.

RSA app or dongle: Something you **have**

ACCESS CONTROL MATRIX

If we can somehow name all the resources (like with file system pathnames), and can describe all the actors (like with the user-id), we can organize this as a matrix.

Consider these requests:

Who	What	When	How
Ken	/usr/ken/fast-wc.cpp	Thursday Aug 27, 10:21.351AM	Open for read and writes
Alicia	/etc/hosts	Tuesday Aug 25, 9:11.112PM	Open for reads
Sagar	…		

NOW SUPPOSE WE HAD THIS TABLE

User	/etc/hosts	/usr/ken/fast-wc.cpp	/dev/proc/pid-781	…
Ken	no access	r/w	r/w	
Alicia	r	no access	no access	
Sagar as “su”	r/w/x	r/w/x	r/w/x	
…				

NOW ALL WE DO IS CHECK THE TABLE AS EACH ACTION IS REQUESTED

Within the Linux system, we can understand many forms of checking as working in this way.

The file system imposes such checks, but Linux also imposes them when process A requests to send a signal to process B

The tables do not "exist" but they allow us to model behavior.

ACCESS CONTROL *LISTS*

Linux only allows objects to have a single owner, and permissions are expressed for the owner, the group and the world.

For a long time there was pressure to extend these into lists. And object could have one owner, but perhaps have different permissions for different users and groups.

Windows still uses this approach, but it never became popular in Linux.

CAPABILITIES

This idea emerged in the 1970's and become popular in the 1980's. For a while there was even competition to offer the best hardware support possible for it.

A capability is a pointer to an object but annotated with the operations the holder is permitted to do on that object.

CAPABILITIES

One home might have a key for the front door, a key for the mailbox, and a key for the bicycle storage shed.

The kernel would allow the creator of an object to also obtain capabilities on it. Like a set of keys

Then the creator can hand out these capabilities, limiting the permissions as appropriate.

For example, "Line printer 6, please print this object: xxx". xxx would be a capability that only permits reading.

CAPABILITIES

A home is an object.

Capabilities are like keys, accessing distinct features.

One home might have a key for the front door, a key for the mailbox, and a key for the bicycle storage shed.

The kernel would allow the creator of an object to also obtain capabilities on it. Like a set of keys

Then the creator can hand out these capabilities, limiting the permissions as appropriate.

For example, “Line printer 6, please print this object: xxx”. xxx would be a capability that only permits reading.

CAPABILITIES

The operations could be any methods the objects support.

So, if an object has a print method, and Alicia has a capability permitting her to invoke print, and a capability on a printer, she could print this object on that printer.

Support for capabilities involves a mix of hardware, kernel and application software features.

REFERENCE MONITORS

This is a concept that starts with a formal model of how a system should behave.

The **reference monitor** has some form of **control** over all accesses to **resources** (including devices, process-to-process interactions).

It validates each action against the model and blocks any that would violate **model constraints**.

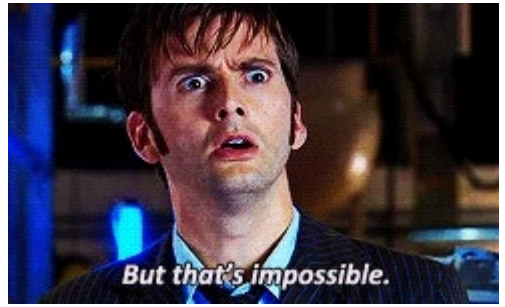
GENERALIZATION OF OTHER APPROACHES

A reference monitor is an abstraction, powerful enough to cover all the concrete options we've mentioned.

But the issue then arises of whether or not it can be implemented.

A model that only covers reads and writes to files can be described in terms of a reference monitor implemented by the Linux file system

… BUT



The 10th Dr. Who:
"But that's impossible."

We could also describe models that confront the reference monitor with undecidable problems!

This is because type checking is undecidable and one could view runtime type checking as a case of a reference monitor!

The mathematical model is very powerful… perhaps *too* powerful! It lets us model policies that are impossible to implement.

EXAMPLE: INFORMATION FLOW MONITORING

This generalizes the idea of sending data and looks at the idea of protecting *information* rather than just data.

The distinction makes sense if you think about how one form of information can sometimes be hidden in another, like hiding one image in the "noise" behind some other image (steganography)

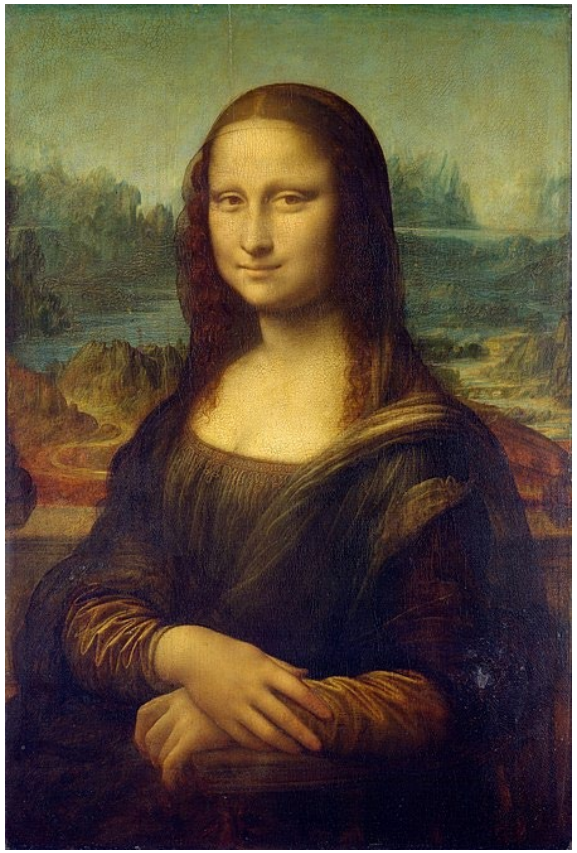
For example think about the least-significant-bits in a color image

WOULD YOU NOTICE TINY CHANGES?

The low-order bit carries very limit visual information.

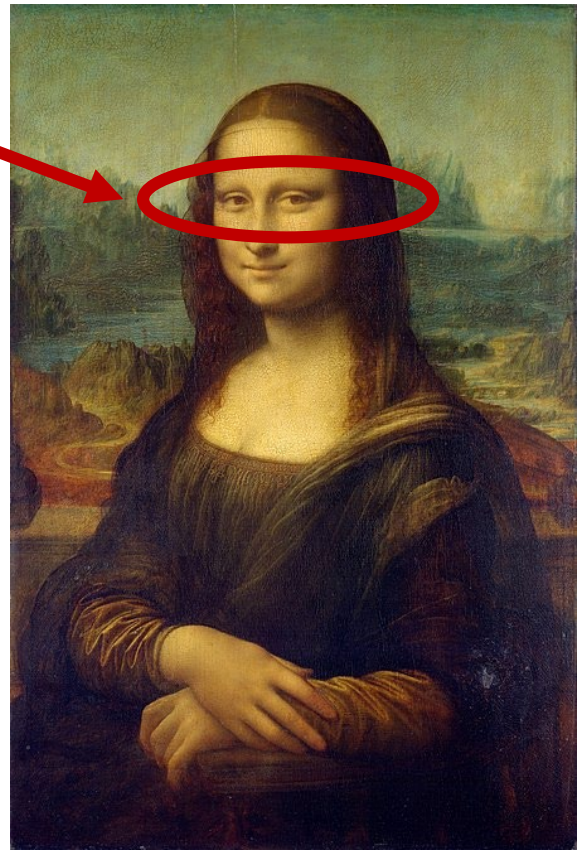
In fact, it is even hard to judge whether it is correct or incorrect

One could use this to create a covert information channel



WOULD YOU NOTICE TINY CHANGES?

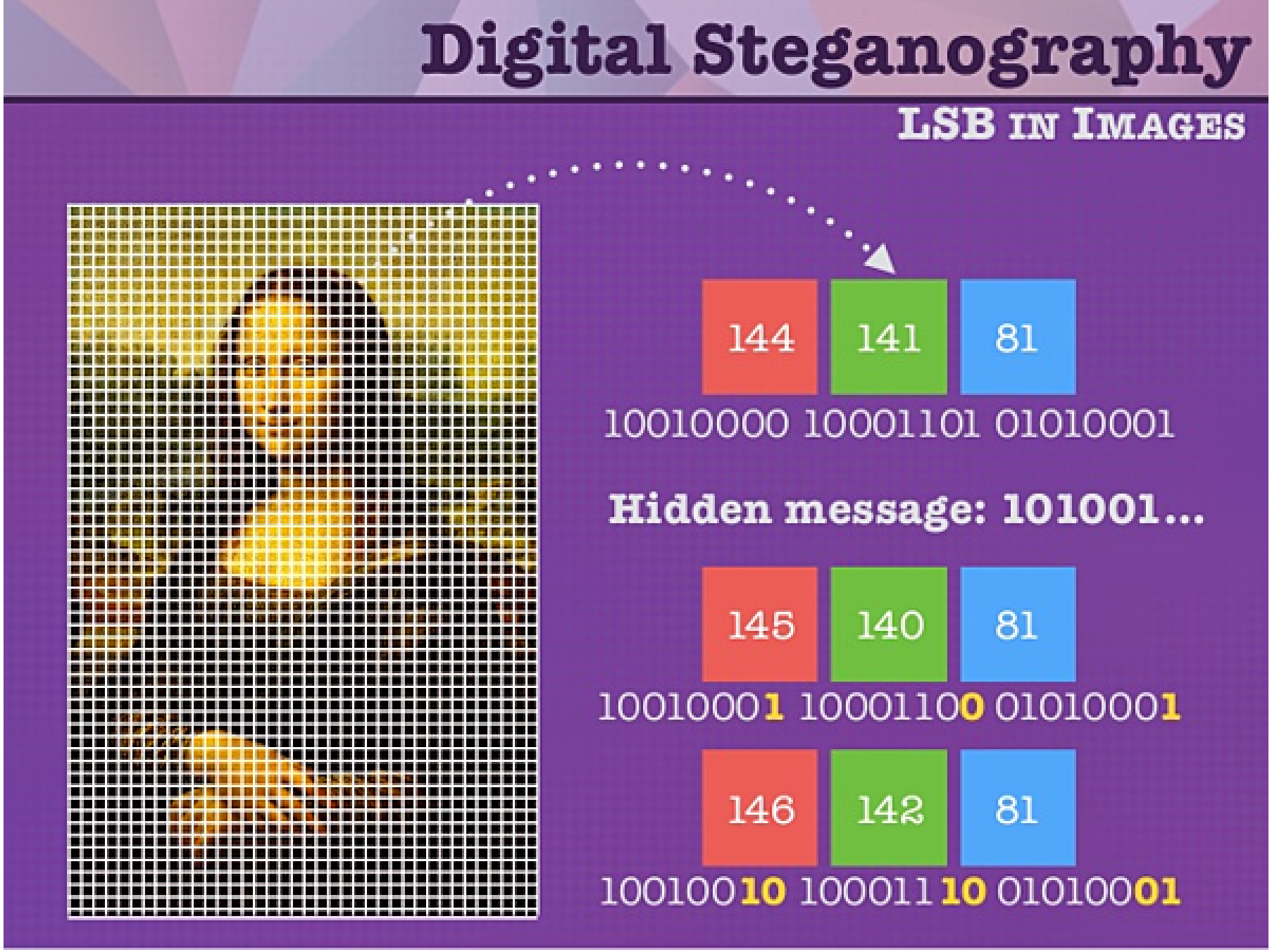
Did DaVinci hide a tiny message in Mona Lisa's eyes? Some believe he did



The low-order bit carries very limit visual information.

In fact, it is even hard to judge whether it is correct or incorrect

One could use this to create a covert information channel



https://www.switchfast.com/blog/threats-hiding-in-plain-sight-digital-steganography-on-the-rise

INFORMATION FLOW MONITORING

As a concept, similar to the idea of reference monitoring.

Mathematical model captures notion of information, as distinct from where that information might currently be.

Then explores ways to limit leakage, or to limit*rate* of leakage via both explicit and covert channels.

THE PROBLEM IS THAT INFORMATION FLOW TRACKING IS HARD!

In normal Linux systems we have pretty blunt "tools" we can use to implement our policies, like firewalls.

These might accomplish a fancy goal like information flow protection: if you can't send images, you can't hide information in them. But this would be very crude.

HOW DO LINUX PERMISSIONS "STACK UP"?

Fancier ideas like capabilities can be implemented in Linux, and this was done by a project called Mach at CMU in the 1990's.

But in fact the approach did not gain much market excitement and eventually was abandoned.

Today Linux simply has a lot of mechanisms, but no explicit access control models or matrix.

HOW DO LINUX PERMISSIONS "STACK UP"?

But Linux has absolutely nothing for fancy tasks like information flow monitoring, or reference monitoring at object granularity

The kernel just treats all data as bytes

THERE ARE TOOLS THAT WILL VISUALIZE THE ACCESS-CONTROL MATRIX FOR YOU

These tools study a Linux system and construct an access control matrix from all the information that is relevant to access.

Then they let the viewer see the data and even change it.

Very valuable in corporate settings where security is important!

BUT THIS IS FAR FROM OUR REAL GOAL

Recall that Tammy wanted to briefly authorize me to get paper.

How does that map to our understanding, and how well can Linux support this?

ATTESTATION: "CORNELL GAVE ME THIS ID"

You arrive at Gates Hall and show an id to the scanner at the door (it runs Linux!)

It checks to see that you have permission, then unlocks the door.

But why should it trust this id? *Cornell-issued ids are trusted by the scanner. This is a form of* ***attestation***. ***Cornell attests for you***.

CHALLENGE: "WHO" IS CORNELL?

Any large organization has some sort of root of decision-making control, like the CEO.

This role delegates various sub-roles to other people. For example, HR signs employment contracts (and lays people off).

Roles are dynamic: people gain and lose roles & authorizations

DOES RSA LET US IMPLEMENT SUCH POLICIES?

A tool like RSA lets individuals sign statements, like "I have hired Janet Smith as our new director of marketing", but these words have no real "logical definitions" unless we provide more detail

Ultimately we are forced to create an entity and roles and authorizations description for the entire organization, and then it evolves as people and roles and tasks evolve – almost like a software description of the entire company and every task.

… YET EVEN IF WE TRIED, WE WON'T BE ABLE TO DO THIS!

First, most companies are much more relaxed about the rules than you might expect (even military or government agencies).

The rules may not really be known or suitable for writing down.

And we often find that even for simple cases, you need the entire database of rules to evaluate even trivial questions!

AUTHORIZATION: "YOU WANT TO DO *WHAT*?"

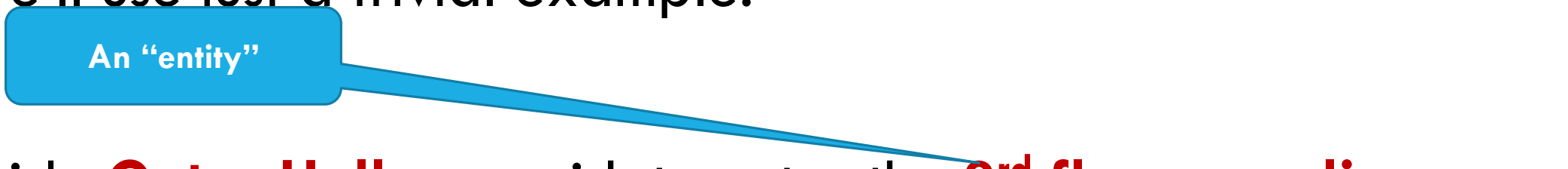
We'll use just a trivial example:

Inside Gates Hall, you wish to enter the 3rd floor supplies room where printer paper is stored. Supplies are tracked on a log.

As a student, you do have permission to be in the building, yet are not *authorized* to sign out printer supplies.

AUTHORIZATION: "YOU WANT TO DO *WHAT*?"

We'll use just a trivial example:

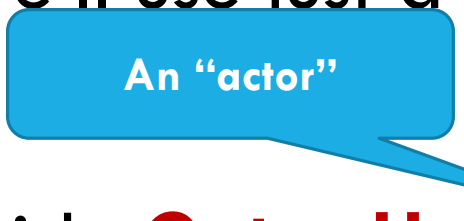


Inside **Gates Hall**, you wish to enter the **3rd floor supplies room** to get printer paper. Supplies are tracked on a log.

As a student, you do have permission to be in the building, yet are not *authorized* to sign out printer supplies.

AUTHORIZATION: "YOU WANT TO DO *WHAT*?"

We'll use just a trivial example:



Inside **Gates Hall**, **you** wish to enter the **3rd floor supplies room** to get printer paper. Supplies are tracked on a log.

As a student, you do have permission to be in the building, yet are not *authorized* to sign out printer supplies.

AUTHORIZATION: "YOU WANT TO DO *WHAT*?"

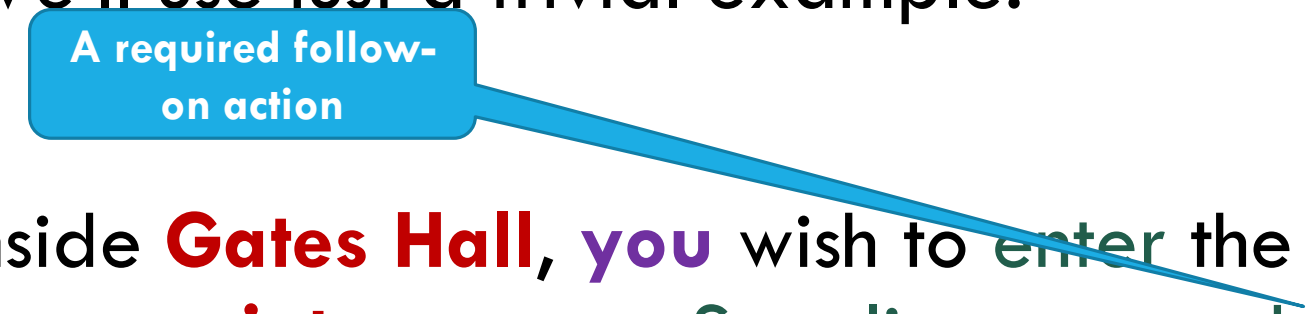
We'll use just a trivial example:

Inside **Gates Hall**, **you** wish to enter the **3rd floor supplies room** to get **printer paper**. Supplies are tracked on a log.

As a student, you do have permission to be in the building, yet are not *authorized* to sign out printer supplies.

AUTHORIZATION: "YOU WANT TO DO *WHAT*?"

We'll use just a trivial example:



Inside **Gates Hall**, **you** wish to ~~enter~~ the **3rd floor supplies room** to get **printer paper**. Supplies are tracked on a log.

As a student, you do have permission to be in the building, yet are not *authorized* to sign out printer supplies.

NOW WE CAN ASK: HOW DO YOU PROVE THAT YOU ARE AUTHORIZED TO DO THIS?

As a student, you need to have been given special permission.

Once you have that permission, and can prove it (and for a limited period of time), you are allowed to perform actions A and B (enter room, take paper) provided you also do C (log it).

DELEGATION: "TAMMY SENT ME FOR PRINTER PAPER"

The term **delegation** is used if someone who has a right to delegate a particular form of authorization gives some other agent permission to act "on behalf" of them.

Perhaps you work for Tammy, and she asks you to fetch paper. She is delegating that task to you, and you are now authorized to get paper on behalf of Tammy.

AUGMENTATION: "NORMALLY, I CAN'T, BUT I'VE OBTAINED SPECIAL PERMISSION"

Delegation gives you a right you didn't have, but you are acting on behalf of Tammy.

In contrast, we say that a right has been augmented if you personally gain that right (perhaps temporarily).

The process acts on its own.

RSA ONLY HELPS A LITTLE

If we write this down in logic (the "BAN" logic is the obvious choice: Burrows, Abbadi and Needham), we could get the active entities to issue credentials that can be signed with RSA

Then the door to the room could check that you have the needed permission and that it was issued by an entity with permission to issue that form of authorization… and the door stays locked, otherwise!

EXAMPLE SEEN IN LINUX

Normally, a process cannot issue operations to the GPU.

But when Linux authorizes a process to use the GPU, it also augments the process to have a memory segment shared with the GPU, and to control the GPU through that segment.

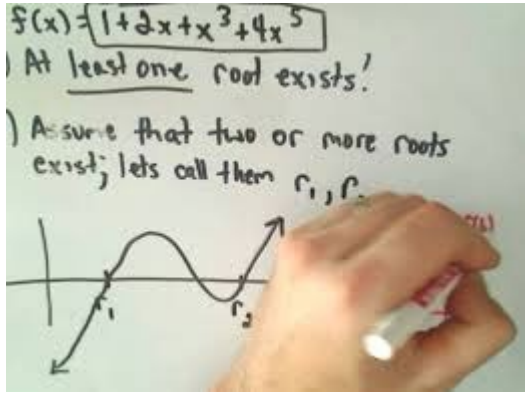
TEMPORARY AUGMENTATION: "I'LL ALLOW YOU IN, BUT ONLY FOR THAT ONE TASK"

When using a library, like a GPU library, the library might be more trusted than the user process.

Here, rights are augmented while inside the library, but revert (like popping from a stack!) when the call returns.

Andrew Myers has done a lot of research on this idea.

"ARE YOU A BOT?"



Some systems demand a *proof of work* as part of authentication. The goal is to filter bots out.

For example, "If you want to connect with me, first solve this homework problem."

To carry out a SYN attack the attacker would need to solve one puzzle for each attempted connection!

GRANULARITY: "JUST BECAUSE YOU CAN DO THAT DOESN'T MEAN YOU CAN DO THIS!"

Our GPU is actually shared with other processes.

Being allowed to access it for one computation doesn't mean you can disrupt those other users, or spy on them.

So, augmented permissions always have some granularity or scope within which they apply.

ROLES: "IN MY ROLE AS DEAN, I AUTHORIZE YOU TO …"

Many systems distinguish who you are from the roles you play.

Think of Kavita Bala. She was a professor, but then became department chair, and now she is dean of CIS.

She is the same *individual*, but in different *roles*. As dean she can authorize things a professor cannot, like signing contracts. Our logic will need to have a way to model this (it is hard!)

REPUDIATION: "I DENY THAT I SAID THAT"

In many systems we worry that a person (or a process they run!) will perform some action but later, the person will claim they did not perform it.

"I didn't fool around with the engine controls and cause it to overheat!"

This is "repudiation". We often want audit trails that give us a way to prove that A did X, and prevent A from repudiating their actions.

TAMPERING: "REALLY? CHECK THE LOGS… "

With superuser permissions, a hacker might try to tamper with logs.

This is why **blockchain** has become important in modern systems. A blockchain entangles records, so that changing anything requires regenerating the entire chain.

Another option is to put the audit log on a write-once medium, so it can't be erased.

FORMALIZING ROLE-BASED SECURITY

Several Cornell researchers have studied was to express these security concepts mathematically.

Perhaps by delegating permission to you to get line printer paper, Tammy unknowingly gave you some other permission…

Formal analytic tools allow systems to evaluate the consequences of actions, perhaps revealing loopholes, contradictions, etc.

HOW EFFECTIVE IS LINUX?

The basic, simple Linux mechanisms work fairly well.

Modern systems often run applications within containerized environments that impose "personal" firewalls and other limits, and these use Linux to gain strong protections.

Kubnetes is an example of a tool for creating containers of this kind, and managing them.

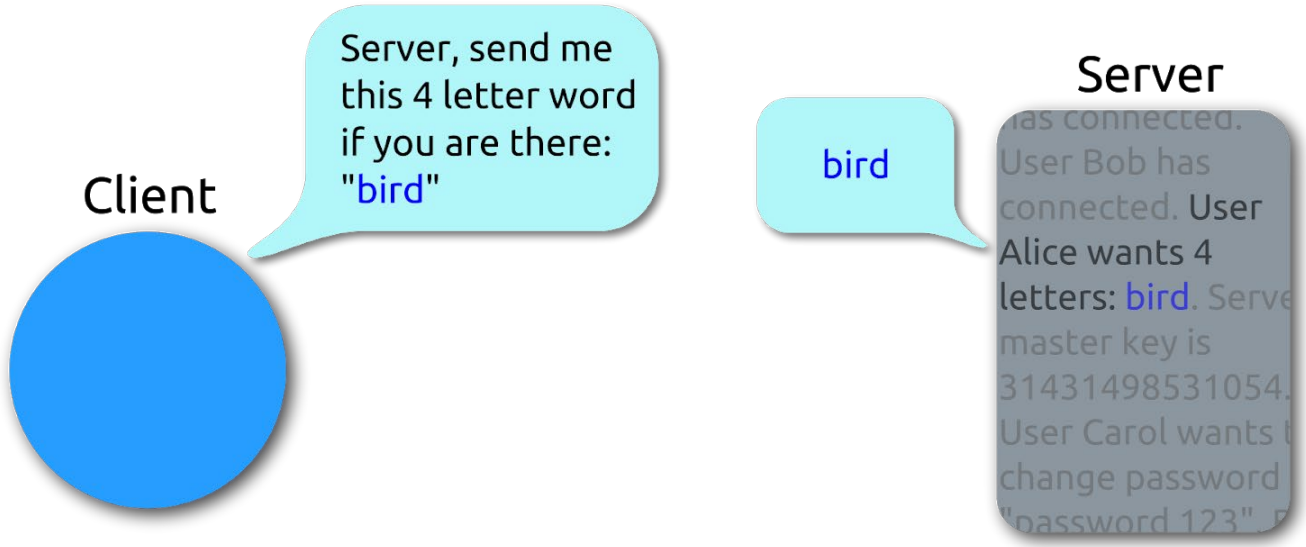
HIGHLY VISIBLE EXPLOITS

Even so, hackers find a way.

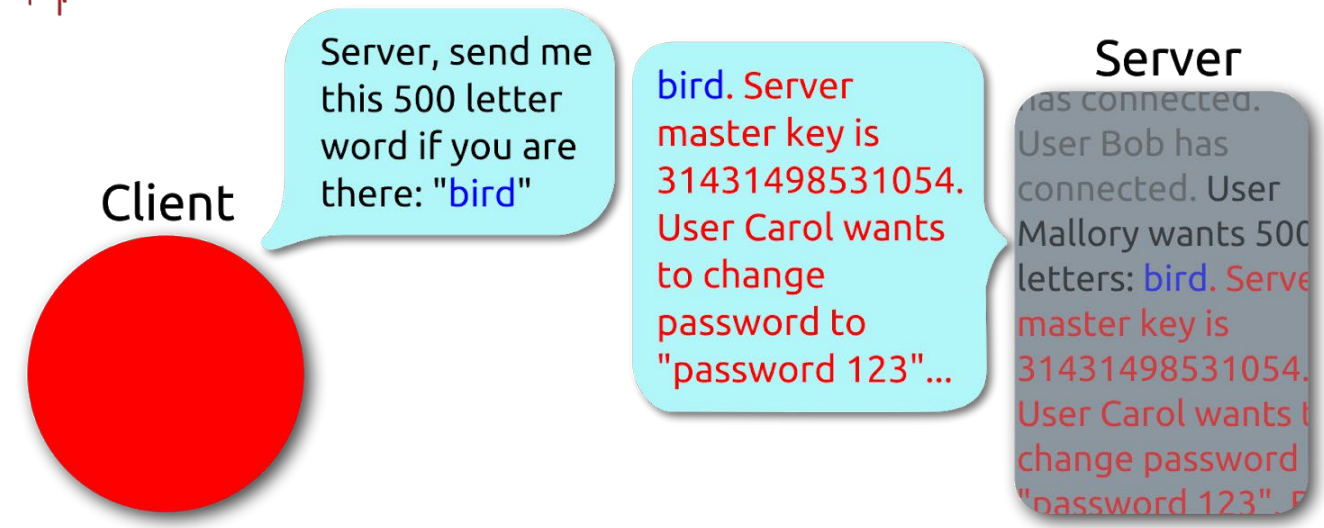
In recent years, hackers discovered that the Linux SSL security code kept some data in a kernel memory area.

They found an exploit that could read this area.

Heartbeat – Normal usage



Heartbeat – Malicious usage



Heartbleed exploit against SSL heartbeat feature

INTEL CACHE ATTACKS

Meltdown

Spectre

Hackers discovered that Intel processors prefetch data even before permissions on the memory segments have been checked.

They configured a process to ignore segmentation faults. Doing so gave them a tiny window in which they could peek at data that the process should not be able to see. (The "peek" step was tricky)

Successfully able to extract security keys from protected code.

BUT EVEN SO, THE PROGRESS IS VERY ENCOURAGING!

In that hospital we mentioned on slide 3, or in an air traffic control system, we really can specify access control policies and enforce them in sensible ways.

The hard cases, today, are mostly seen when *organizations* need to cooperate (like at the Cornell/NYU/Sloan tri-institutional medical center complex). Use of blockchains for audit trails is a very promising remedy that may help in such settings.

BASICALLY, ANOTHER "SWISS CHEESE" STORY

We have powerful conceptual abstractions and tools

We use mixtures of these: in one situation, file system protections. In a different one, two factor authentication with digital certificates. A third situation requires a firewall

Individually none is adequate. But jointly, they can solve our need

SUMMARY: AN ARMS RACE! VERY HARD PROBLEMS… BUT WE "MIGHT NOT LOSE"

The theory of access control and information flow is powerful, but the basic Linux and C++ features are limited and favor speed over security coverage.

There have been many efforts to strengthen Linux and C++, but they added costs and complexity and were ultimately rejected.

Modern security tools focus on using analysis to discover risks.