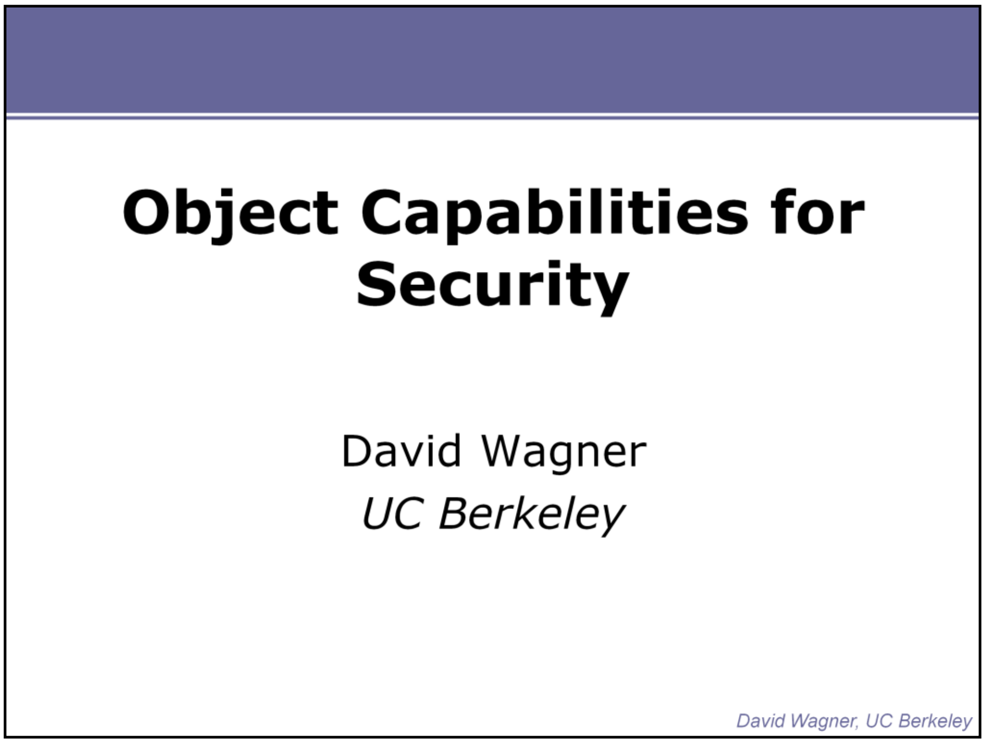


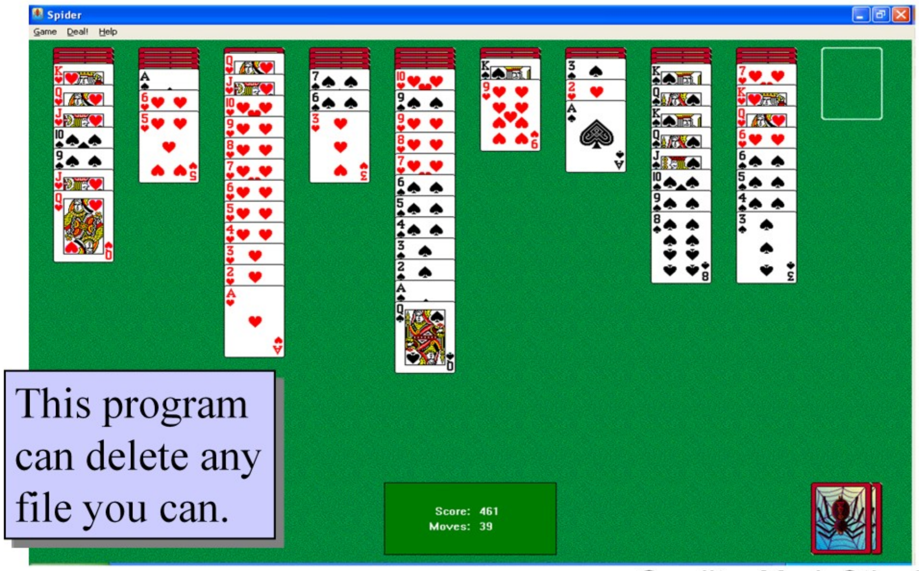
Object Capabilities for Security

David Wagner
UC Berkeley

David Wagner, UC Berkeley

I'd like to use this presentation to tell you about an approach to architecting secure systems that you might not be familiar with. It goes by the name "object capabilities", and the goal is to help us do a better job of following the principle of least privilege and of building secure systems. There is a small community of folks who have been quietly, over the past decade or so, working out the details of how to do this. I've been studying their work for the past several years, and they've got some interesting insights that I'd like to share with you, ideas don't seem to be very widely known. I should disclose up front that I can't take any credit for any of these ideas; most everything I show you in this talk will be someone else's ideas.

A very powerful program



This program can delete any file you can.

Credits: Mark Stiegler

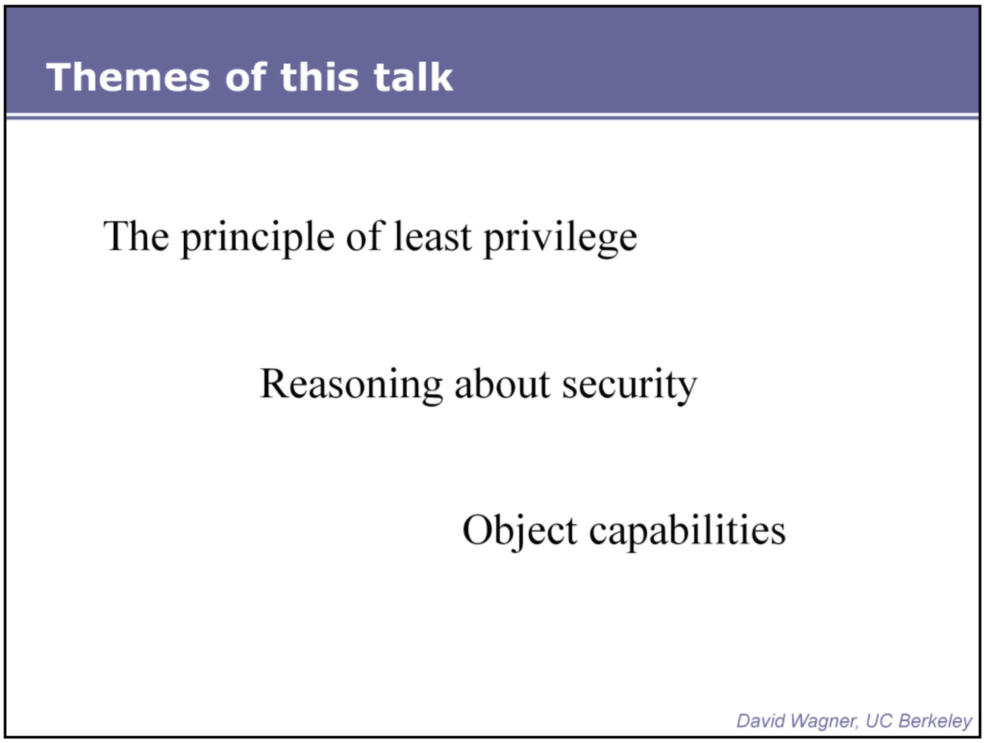
Let me start off by showing you a very powerful program - a program that can delete all your files - a program that can scan your email for interesting tidbits and sell them on eBay to the highest bidder. Many of you know people who spend hours running this program. What is this program? * Solitaire -- and every other program you use. *

The only rights Solitaire really needs to do its job are the ability to write in its window and to receive UI events directed at its window. Yet -- like every other program you execute -- Solitaire runs with a lot more rights than that. It runs with all of your authority.

Power is dangerous. While Solitaire probably doesn't do any of those nasty things, if Solitaire became corrupted by a virus, it could. This is why viruses are able to cause so much damage.

You may recognize this as a violation of the principle of least privilege. Three decades ago Saltzer and Schroeder taught us that the less power you give to a program, the less harm it can do when it runs beserk. Unfortunately, we don't seem to have learned the lesson very well. On today's systems, programs typically run with all the privileges and rights of the user who invoked them. For instance, when you play Solitaire on your computer, the Solitaire app has full access to all of your files. It can delete any file you can. But, of course, it doesn't need all those powers. In general, giving every program you run access to all your files is dangerous, because if any of those programs get compromised, you can kiss all your files goodbye -- they're toast. That's why a single virus or worm or buffer overrun vulnerability can do so much damage, on today's platforms.

The solution is obvious: rather than granting Solitaire the authority to destroy your life, we should only grant Solitaire the rights it needs to do its job, and no more. The same goes for every other application we run. The goal of object capabilities is to make that easier.



Themes of this talk

The principle of least privilege

Reasoning about security

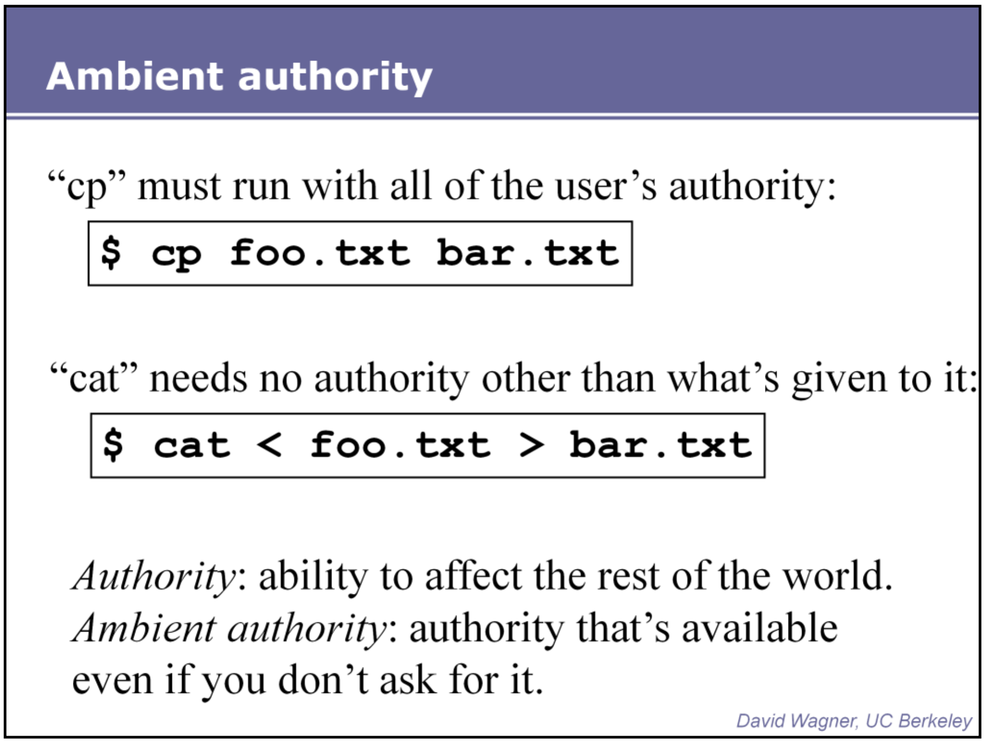
Object capabilities

David Wagner, UC Berkeley

So, in this talk, I'm going to talk a lot about the principle of least privilege.

Also, if you want programmers to minimize the privileges they use, we have to give them some way to reason about how their code manipulates privileges. So, * I also want to talk about ways to help programmers reason about security. These are the two primary goals of * the object capabilities paradigm.

Let's start with the principle of least privilege.


Ambient authority

"cp" must run with all of the user's authority:

```
$ cp foo.txt bar.txt
```

"cat" needs no authority other than what's given to it:

```
$ cat < foo.txt > bar.txt
```

Authority: ability to affect the rest of the world.
Ambient authority: authority that's available even if you don't ask for it.

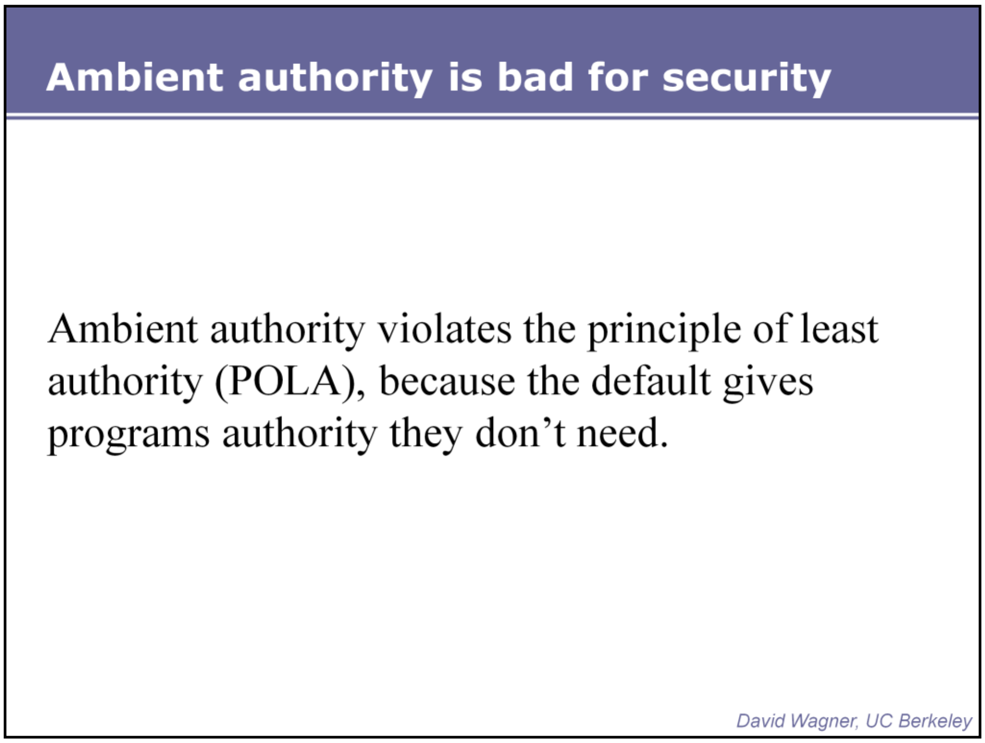
David Wagner, UC Berkeley

One of the deadliest enemies of the principle of least privilege is a pattern that I'll call ambient authority. Suppose we want to make a copy of the file foo.txt. The traditional way to do that, in Unix, is to use the "cp" command. When you run "cp", it inherits all of your powers -- including the power to access all your files. If you think about it, it has to. The interface to "cp" is that it receives two strings -- two filenames -- and it has to be able to open the files they designate. If you want to be able to use "cp" on any of your files, without restriction, you have to give "cp" the power to read and write all your files. Consequently, all the code in the "cp" command is running with the full privileges of the invoking user, without restrictions; it's as if those privileges were "in the air, for the taking".

* Compare to an alternative way of copying the file. In the second example, "cat" does not need the power to open any files on its own; instead it receives file descriptors for the source and destination file, pre-opened by the shell, and that's all it needs. Thus, the program "cat" could be run with no special powers of its own, and without the ability to open any files. Today's systems don't work that way at the moment, but they could. If they did, then the code of "cat" would not have access to any of your files, unless you explicitly take some step to pass it an already-opened file descriptor to that file. That would be much better, from the point of view of the principle of least privilege. The difference between the two is that "cp" is written to use ambient authority, whereas "cat" (as used here) does not need ambient authority.

* By now you may be wondering where the term "ambient authority" came from. Well, that's the term the object capabilities community uses for this pattern of insecurity. Because my objective in this talk is to share with you some of what I learned from that community, I'm going to try to use the same language they use. I should warn you that they've accumulated quite a few terms that might be unfamiliar to you, and they're going to appear throughout the talk, so bear with me. The first of these is "authority", which is an attempt to clarify some of the fuzziness surrounding what we sometimes call privileges, powers, rights, etc. The authority a program has is exactly the set of ways that the program can causally influence the outside world -- and that includes both direct influence and indirect influence. For instance, if the program has permission to open the file "bar.txt", then its authority includes the power to read or write the contents of bar.txt. Authority is transitive: if Alice has authority to talk to Bob, and if Bob has authority to modify the file "bar.txt", and if there is some message Alice can send to Bob that will cause him to modify "bar.txt", then Alice also has authority to modify "bar.txt" -- and that's true even if Alice doesn't have permission to directly modify that file on her own.

That's the definition of "authority". Now, "ambient authority" is authority that is floating in the air. In Unix, just by virtue of running under Alice's userid, a program is granted authority to access all of Alice's files. It's as if the program is implicitly provided with a "big bag of rights", and when a program tries to open a file, the OS checks to see whether any of the rights in that bag suffice to allow the open operation to proceed. The program doesn't need to explicitly identify under what authority it claims the right to open that file -- that's left implicit.



Ambient authority is bad for security

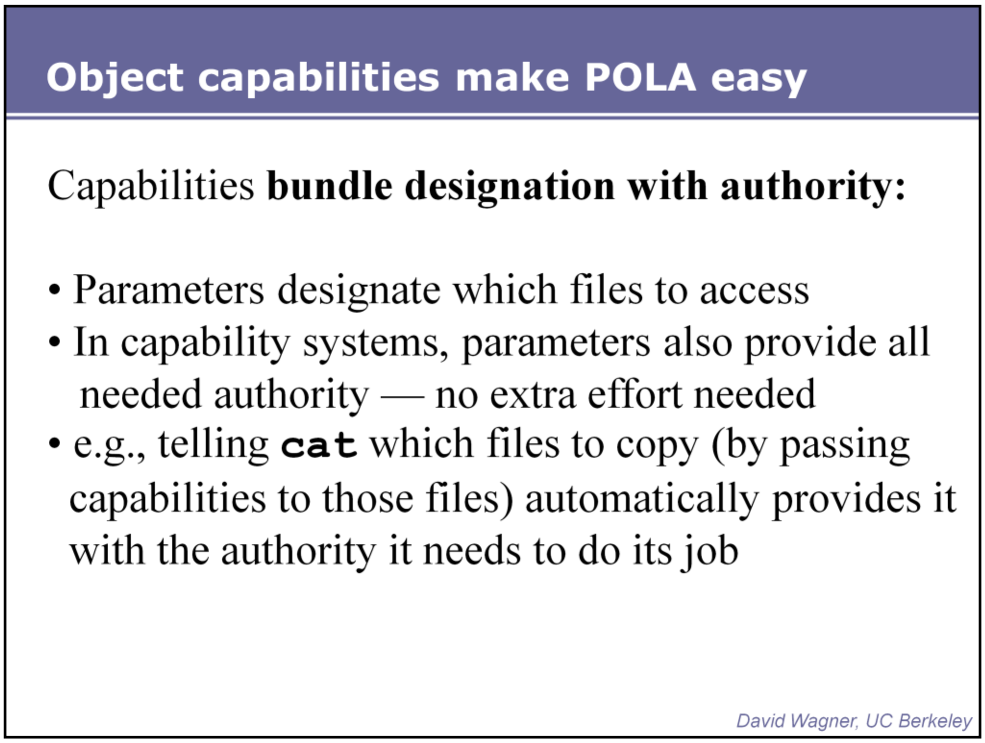
Ambient authority violates the principle of least authority (POLA), because the default gives programs authority they don’t need.

David Wagner, UC Berkeley

The problem with ambient authority is that it gives programs lots of authority by default, whether they need it or not. The result is that, in ambient authority systems, most programs run with far more authority than they need -- which means that if something goes wrong, they can do a huge amount of damage. This is exactly the opposite of what the principle of least privilege tells us to do.

By the way, now that I’ve introduced the concept of authority, I want to be more precise and say that our real goal should be to respect what we might call the principle of least authority: give programs only the minimum *authority* they need to get the job done. Capabilities folks call this POLA for short. So, capability folks might put it this way: ambient authority is bad for security because it violates POLA.

(Are you starting to get the hang of the capability jargon now?)



Object capabilities make POLA easy

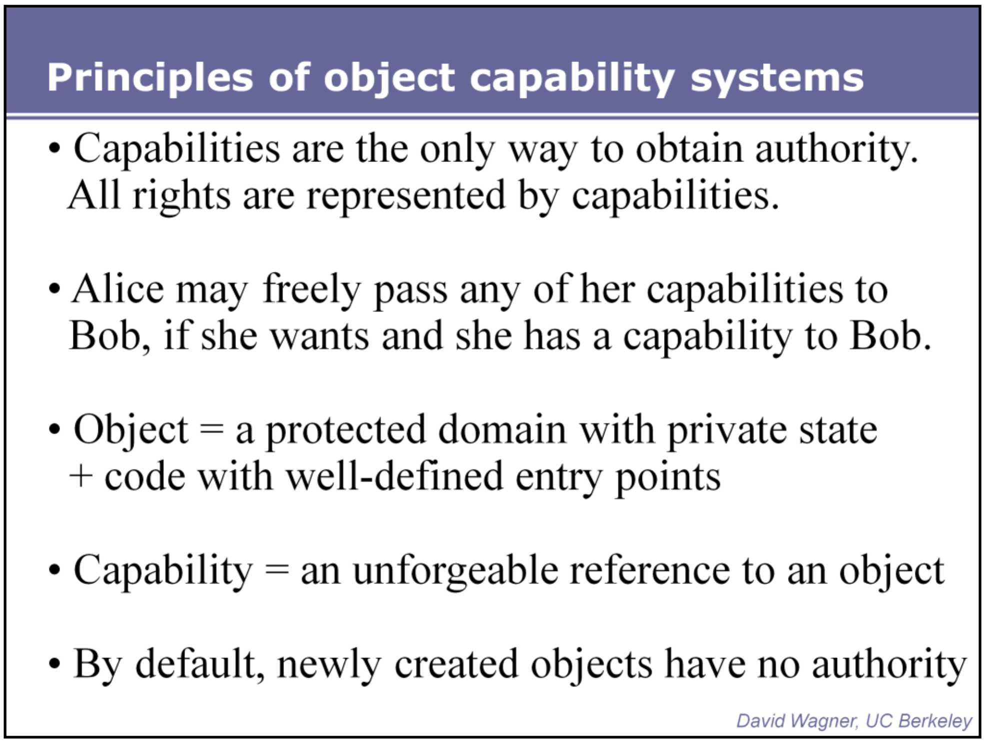
Capabilities **bundle designation with authority:**

- Parameters designate which files to access
- In capability systems, parameters also provide all needed authority — no extra effort needed
- e.g., telling **`cat`** which files to copy (by passing capabilities to those files) automatically provides it with the authority it needs to do its job

David Wagner, UC Berkeley

Object capability systems are designed to make it easy to practice the principle of least authority. The way they do that is by making the process of designating which files to operate on also automatically convey the authority needed to perform that operation. Remember “cp”? There, the two command-line parameters to “cp” identify which files to operate on, but because they’re just strings, the parameters don’t themselves provide the authority to perform the copy operation -- instead, “cp” needs to get that authority from its ambient authority. This means that “cp” ends up getting a lot more ambient authority than it needs on any particular invocation -- it receives the right to modify a whole bunch of files it doesn’t even know anything about, and that’s overkill. In contrast, with “cat”, the mechanism that users use to tell “cat” which files to operate on also provides “cat” with the rights it needs to perform the copy operation. That’s because, to use “cat”, you pass it two capabilities to the files you want to operate on -- two file descriptors, and file descriptors are the quintessential example of a capability. In short, the capability way is to do it like “cat” does -- pass capabilities, not filenames.

The capability way has two advantages. First, the least authority that cat needs (on any particular invocation) is a lot less than the least authority that cp needs, so the capability way makes programs more amenable to POLA. Second, in object capability systems, we don’t need to take any special steps to specify the set of rights to give to “cat”. We don’t have to configure complex security policies. Instead, the person who invokes “cat” is implicitly specifying what authority “cat” should receive. This gets us the best of both worlds: it leads to program structures that need a lot less authority than the alternative, and it also provides those programs with exactly that minimal authority and nothing more. Object capabilities make POLA come almost for free if you follow capability principles.



Principles of object capability systems

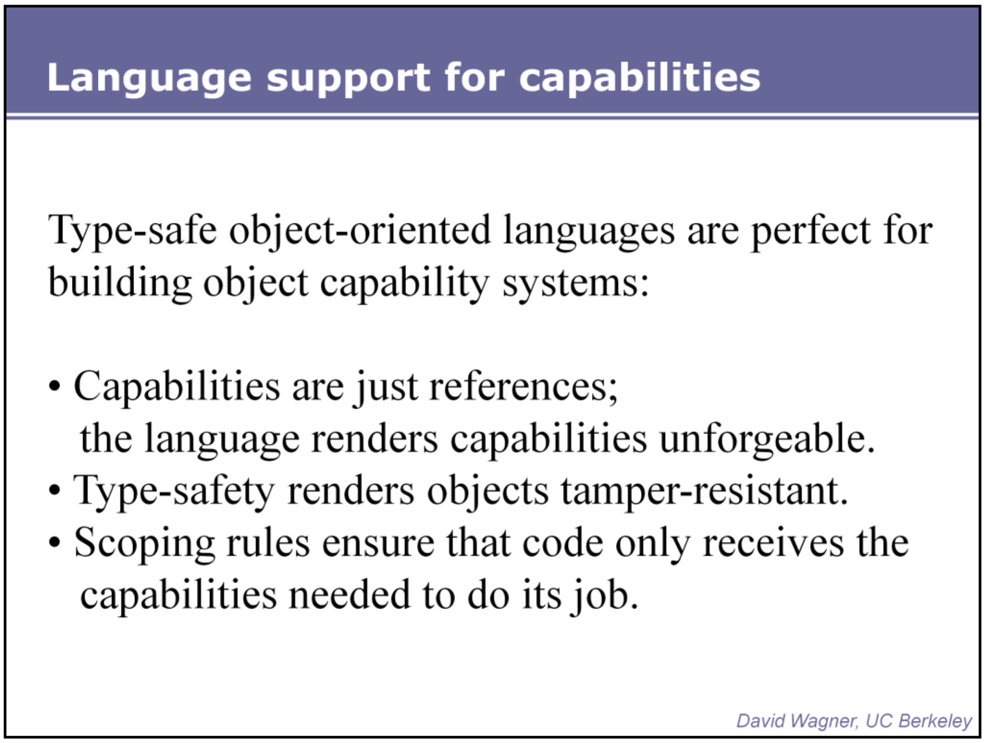
- Capabilities are the only way to obtain authority. All rights are represented by capabilities.
- Alice may freely pass any of her capabilities to Bob, if she wants and she has a capability to Bob.
- Object = a protected domain with private state + code with well-defined entry points
- Capability = an unforgeable reference to an object
- By default, newly created objects have no authority

David Wagner, UC Berkeley

What's an object capability system? A capability system is composed of just objects and capabilities -- nothing more. Every physical resource is represented by an object. Every entity that we want to control access to is represented by an object. Everything that conveys authority -- i.e., provides the ability to affect the outside world -- is represented by an object. Capabilities are unforgeable references to these objects, which can be held by other objects. Think of a graph where objects are the nodes and capabilities are the edges.

A reference to an object -- a capability -- lets you send a message to the object it designates. Capability systems are set up so that the only way you can side-effect the rest of the world is by sending a message via some capability. That means that the only way to influence the outside world is by holding a capability that will let you do so. In other words, in capability systems, capabilities are the sole carriers of authority.

Also part of the concept of capability systems is that objects, when they're created, initially come to life with no capabilities, or only those capabilities explicitly passed to them by their creator. Thus, newly created objects initially have no ability to influence the outside world, other than what their creator provided them with. To put it another way, that means that all entities spring to life with, by default, no authority at all -- which is the exact opposite of ambient authority.



Language support for capabilities

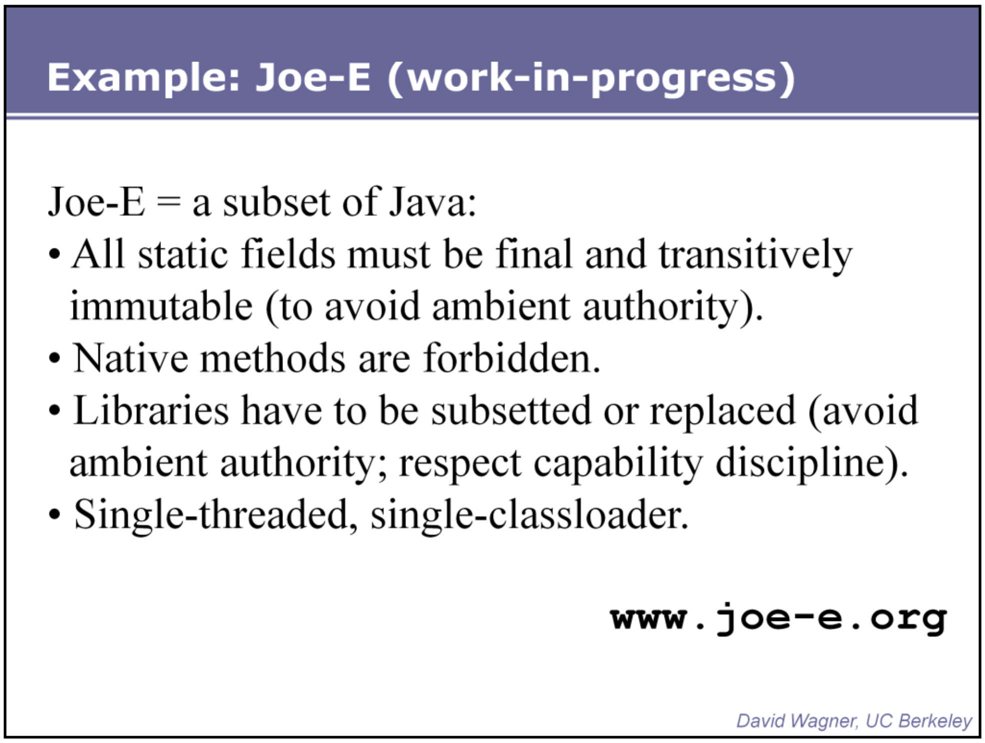
Type-safe object-oriented languages are perfect for building object capability systems:

- Capabilities are just references; the language renders capabilities unforgeable.
- Type-safety renders objects tamper-resistant.
- Scoping rules ensure that code only receives the capabilities needed to do its job.

David Wagner, UC Berkeley

Modern OO languages are a perfect fit for capabilities. Think of a language like Java, for instance. In OO languages, objects may contain both private state (instance variables) and code (methods) that cannot be tampered with by anyone else, thanks to the language's type-checking facilities. Also, in modern OO languages, references are unforgeable, and that's very useful as well. The unforgeability of references makes capabilities unforgeable: the only way to get a reference to some object is to create that object yourself or to be passed a capability to it. In particular, you can't make up capabilities out of thin air. Many OO languages also provide a lot more -- e.g., inheritance, polymorphism, or subtyping -- but we don't need them for our purposes.

Last -- and maybe not so obvious -- the scoping rules of these languages is a good fit for practicing POLA. One of the requirements of a capability system is that we must make sure that the universal scope -- the lexically outermost, the environment available to all code -- provides no authority. That's necessary if we want to avoid ambient authority. With that done, the only way that an object can receive authority is if someone else passes it a capability providing that authority. Notice how this provides an "opt-in" regime, where the default is no authority except what has been explicitly passed through method invocations.



Example: Joe-E (work-in-progress)

Joe-E = a subset of Java:

- All static fields must be final and transitively immutable (to avoid ambient authority).
- Native methods are forbidden.
- Libraries have to be subsetted or replaced (avoid ambient authority; respect capability discipline).
- Single-threaded, single-classloader.

www.joe-e.org

David Wagner, UC Berkeley

For instance, Adrian Mettler and I are building a language called Joe-E. Joe is a language for building capability-based programs, and to try to minimize the learning curve for programmers, Joe-E is specified as a subset of Java. So most of it would be pretty familiar to you. Actually, Java, the language, is already quite close to being an object capability language. Java has only a few features that violate capability rules: static fields are accessible in the universal scope, and have to be restricted to ensure they don't provide authority; native methods fall outside the type-safe core; and Java's thread synchronization and classloaders need to be reconciled with capability style, so we have stripped them out of Joe-E for the time being.

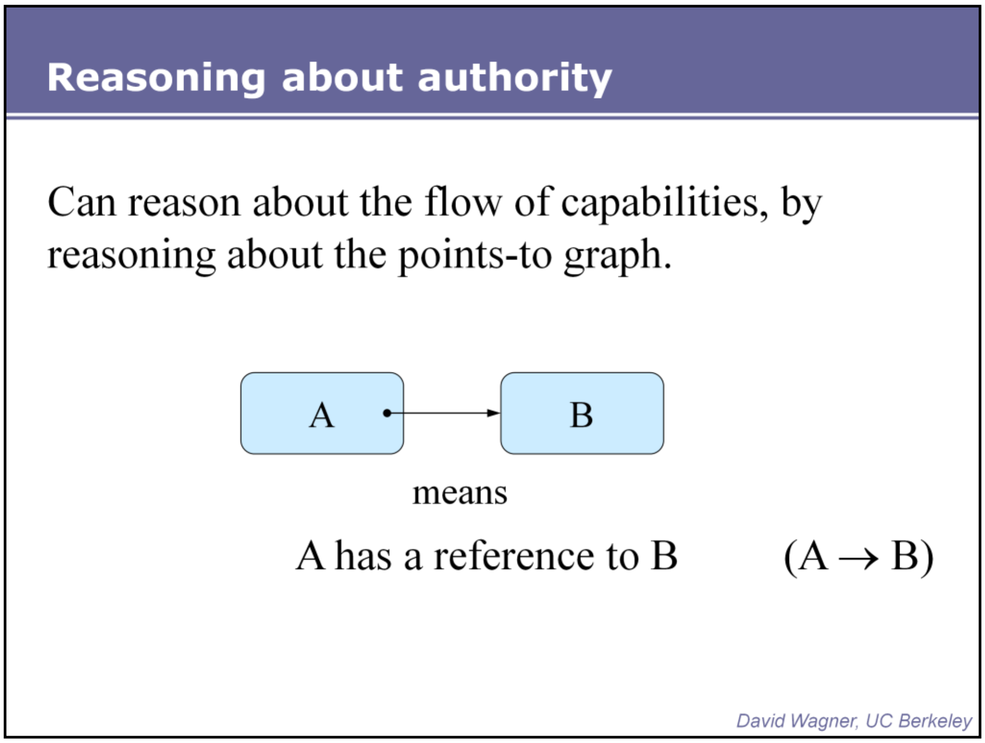
Of course, Java also comes with a huge set of class libraries, which were very important to its success, but many of which were not designed with the capability paradigm in mind. For instance, some portions of the class libraries provide ambient authority, which is a no-no in a capability language. Therefore, we're working on subsetting these libraries to eliminate the parts that are incompatible with object capabilities.

A key property is that Joe-E is a strict subset of Java. Our motto: we're add security to Java, not by adding features to the language, but by removing unsafe features. Because Joe-E is a subset of Java, programmers can continue to use all of their favorite Java tools: profilers, debuggers, development environments, program analyzers, optimizing compilers and VMs, etc.

All of this is just to say that Joe-E is a language that a lot of you would probably find very familiar.

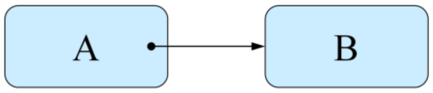
For now, Joe-E is only intended for building new code – not for taking legacy Java code and painlessly "making it secure". You have to build software a bit differently if you want them to follow the object capability paradigm.

Joe-E is inspired by E, an earlier object capability language designed by Mark Miller.



Reasoning about authority

Can reason about the flow of capabilities, by reasoning about the points-to graph.



means

A has a reference to B $(A \rightarrow B)$

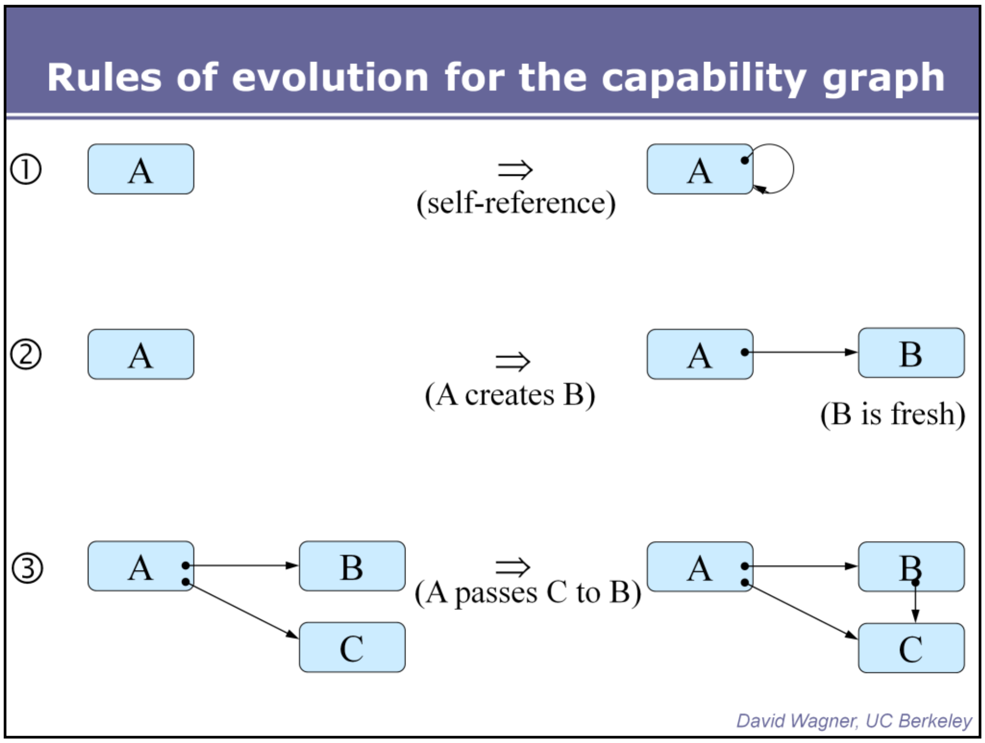
David Wagner, UC Berkeley

Now I want to move on to my second subject: how object capability systems make it easier to reason about the flow of authority in a program. For instance, maybe I want to know what authority some chunk of code might possibly receive, as part of assessing what damage it can do. Or maybe I have an especially precious capability and I want to determine what parts of the code might be able to get their hands on that capability. That's what I mean by reasoning about authority. These are fundamental tasks.

The ability to reason about authority is very useful. For instance, suppose I have a chunk of data stored in memory, and I want to know which parts of the program can write to that chunk of data. With a monolithic C program, there's no way to know, short of inspecting every single line of code. Any C code anywhere in the program could (for all I know) forge a pointer to the chunk of data and start scribbling over its contents. In contrast, in an object capability system, if I create the chunk-o'-data object, I can easily identify the set of modules that might be able to gain access to that object by just following along in the code to see who that object reference is passed to.

One of the nice things about capability systems is that, because capabilities are the sole carrier of authority and capabilities are just references, the job of reasoning about authority should be intuitive to programmers, because it just comes down to reasoning about the flow of references through the code, a task that programmers are used to thinking about.

In particular, one of the key reasoning patterns that I've seen involves reasoning about the graph of object references. Imagine a graph where each object is a node, and where we draw an edge from Alice to Bob if Alice holds a reference (a capability) to Bob. That's the reference graph. Given such a graph, we can identify which objects Alice is currently able to influence by tracing transitively along the edges in the graph, and that reachability computation will tell us which objects Alice can reach in the reference graph.

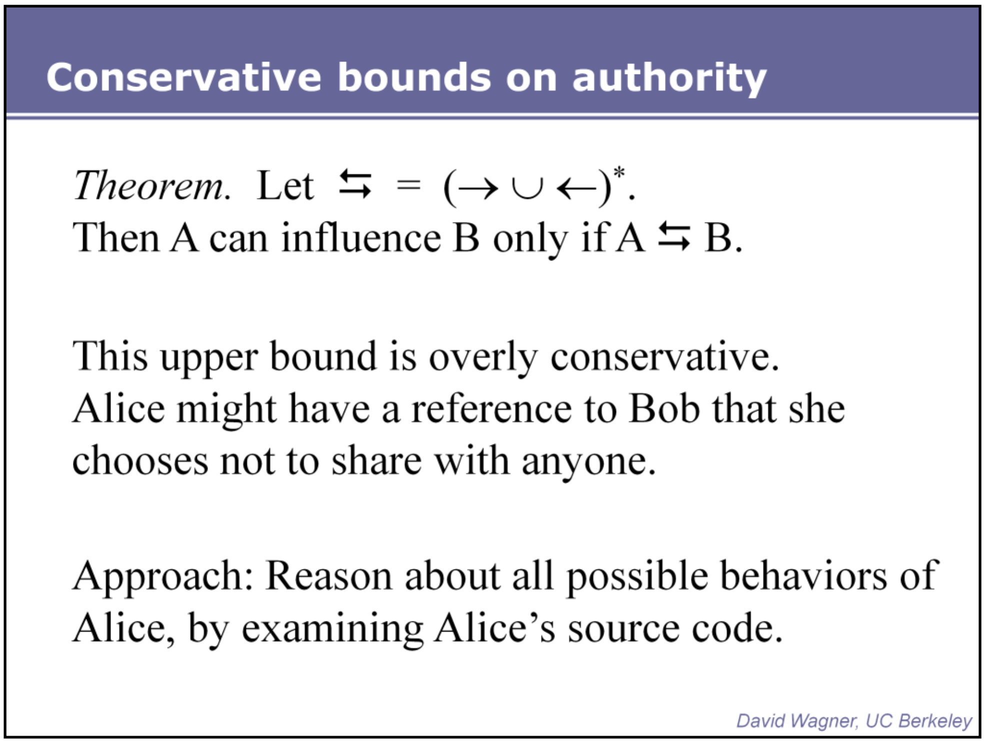


Rules of evolution for the capability graph

Of course, the graph evolves as the program executes. For instance, when you create a new object, this adds a new node to the graph, and gives its creator a reference to the newly created object (I.e., draws an edge from the creator to createe). See rule (2).

I show here the three atomic transformations that can be applied to the capability graph as the program executes. Rule (1) says that any object can get a reference to itself. Rule (2) says that any object may create a fresh new object and get a reference to it. Rule (3) says that if Alice has references to Bob and Carol, and if Alice chooses to share her Carol-reference with Bob, she may do so. The way she does that is by invoking one of Bob's methods and passing the Carol-reference as a parameter to that method. The result is that Bob now has a Carol-reference.

The only way the capability graph can evolve over time is by repeated application of some sequence of these rules.



Conservative bounds on authority

Theorem. Let $\leftrightarrows \; = (\rightarrow \cup \leftarrow)^*$.
Then A can influence B only if A $\leftrightarrows$ B.

This upper bound is overly conservative.
Alice might have a reference to Bob that she chooses not to share with anyone.

Approach: Reason about all possible behaviors of Alice, by examining Alice's source code.

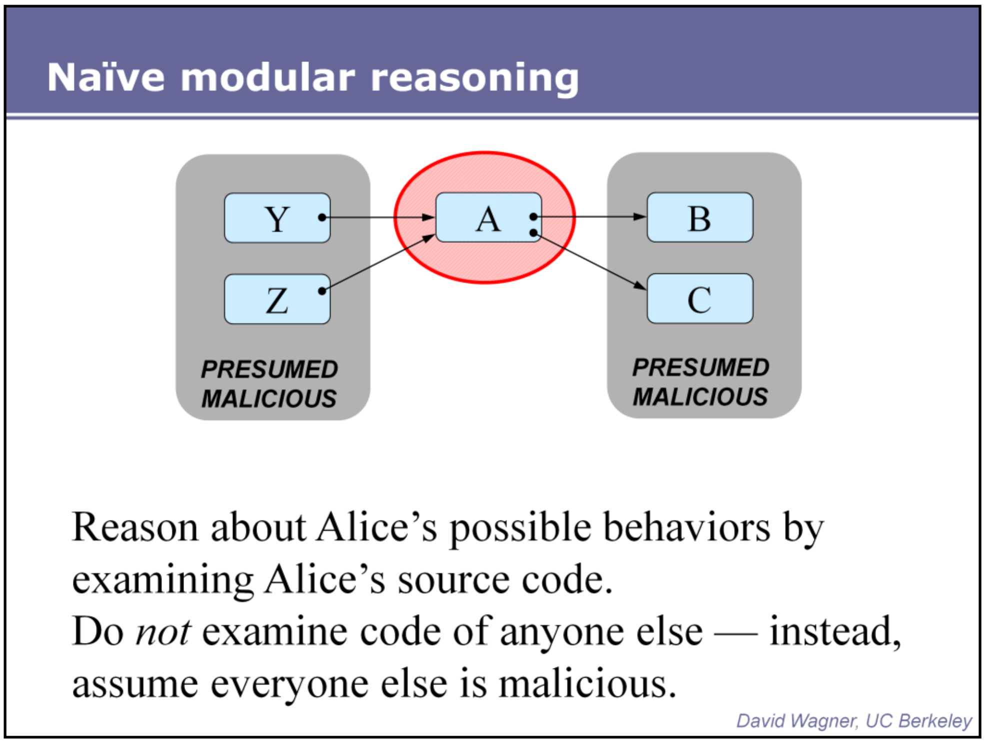
David Wagner, UC Berkeley

This gives us a way to reason not only about what authority Alice might have at this moment, but also what authority Alice might be able to obtain in the future (e.g., by interacting with other objects). In particular, we can use the reference graph to get an upper bound on the set of capabilities that Alice might be able to get her hands on by assuming that every one of the previous three steps is applied whenever it is applicable. That'll give us a conservative upper bound. That leads to the following theorem. Consider the edge-relation associated with the initial configuration of objects, and build the transitive, reflexive, symmetric closure of this relation. If Bob is not reachable from Alice in this extended relation, then the theorem says there is no way for Alice to influence Bob. In other words, if Alice is completely disconnected from Bob in the initial configuration, then they'll remain disconnected forever, no matter how the graph evolves. In slogan form: "Only connectivity begets connectivity."

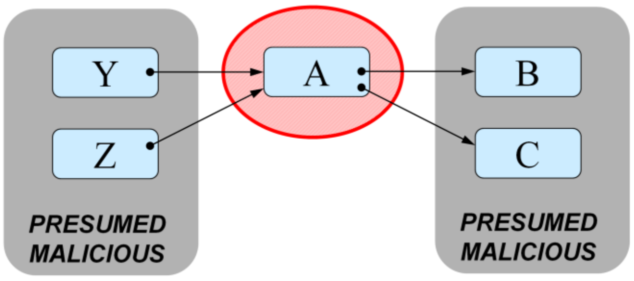
The ability to get an upper bound on the authority that Alice might have by doing a reachability analysis on the code (examining Alice and transitively everyone she is connected to) is very useful. This turns out to be a useful pattern of reasoning about capability code.

However, often the straightforward upper bound obtained by naïve reachability analysis is too conservative. Reachability analysis assumes that all possible graph-transformations that are applicable, will eventually be applied. For instance, if Alice has a reference to Bob, simple reachability analysis starts from the conservative assumption that Alice might share her Bob-reference with everyone she can. That's sound, but in practice that assumption is often too conservative. For instance, we might be able to look at the code of Alice and see that Alice will never share her reference to Bob with anyone else. If so, that lets us cut off the transitive reachability exploration at that point in the graph.

This leads to my next point: if we look at the code of some objects and reason about their possible behaviors, we can often get tighter upper bounds on authority. However, we don't want to go too far down this path. Any pattern of reasoning that requires us to examine the code of every object won't scale. We can't keep that much in our heads at any one time. Therefore, we need patterns of reasoning that allow us to look at the code of a small subset of objects, analyze their possible behaviors, without inspecting the code of anyone else.



Naïve modular reasoning



Reason about Alice's possible behaviors by examining Alice's source code.
Do *not* examine code of anyone else — instead, assume everyone else is malicious.

David Wagner, UC Berkeley

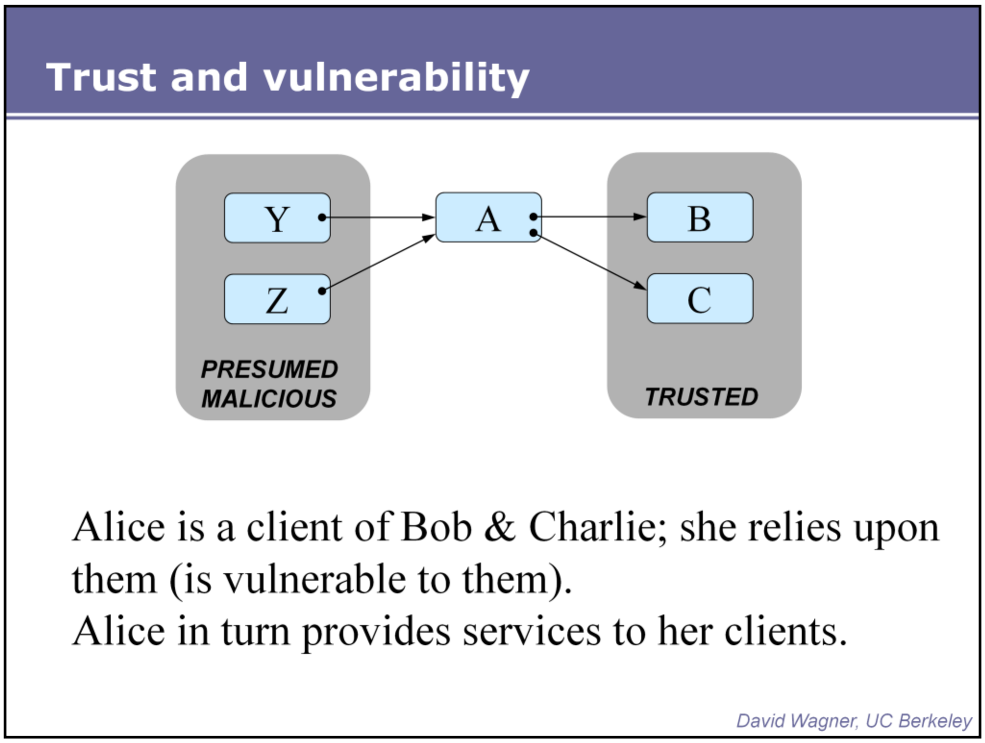
That's the idea behind modular reasoning. The goal of modular reasoning is to enable us to reason about Alice by looking at Alice's source code and analyzing Alice's potential behaviors, without assuming anything about the behaviors of the other objects in the system. One possible way to think about that is to * draw a perimeter around Alice. We inspect the source code of everything inside the perimeter and analyze how it might behave. Anything outside the perimeter is not inspected. We try to prove things that will hold no matter how the stuff outside the security perimeter behaves (subject only to the condition that it must respect the rules of the capability language). If you like, you can think of this as quantifying over all possible implementations of the external objects -- or, you can think of it as treating everything outside the perimeter as malicious and colluding. Either way, we can apply conservative reachability analysis to bound what the external objects can do, and then we can combine that with the set of behaviors for Alice that are consistent with Alice's implementation to see what properties can be guaranteed to hold.

Of course, we can generalize this by drawing a perimeter around a constellation of cooperating objects, instead of drawing the perimeter around just one object.

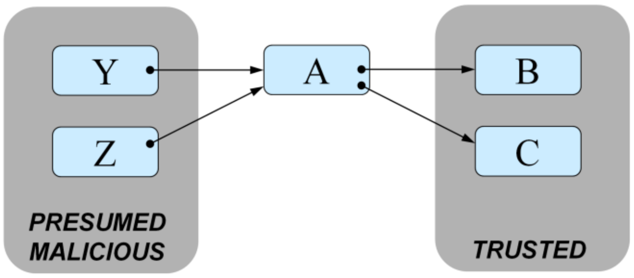
Notice the close connection between modular reasoning in PL and security analysis in capability systems. Modular reasoning involves drawing a perimeter around part of the code and trying to prove properties that will hold over all possible instantiations of the external objects. Security analysis involves drawing a security perimeter around part of the code and trying to prove properties that will hold even when the external objects are all malicious and colluding. Those are the same thing!

Modular reasoning is essential to securing large systems. It is the only means we have for dealing with complexity and scale and the fact that we can only keep so much in our head at one time. The problem with languages like C is that there is no hope for performing modular analysis of a large C application, because the lack of memory-safety means the part of the code we haven't inspected could tamper with the state of the code we have inspected in utterly unpredictable ways and violate all our invariants. If you want to perform a code review of a large C application, the difficulty of performing modular analysis means that you essentially have to keep the whole thing in your head, and that's horrendous to contemplate.

That said, the view of modular reasoning I've shown on this slide is highly naïve. It often doesn't work in practice, because in practice Alice often has to rely on services provided by other objects. For instance, maybe Alice uses a Hashtable to store data; it doesn't make sense to treat the Hashtable as malicious, if Alice has made a conscious decision to rely upon the Hashtable's correct operation.



Trust and vulnerability



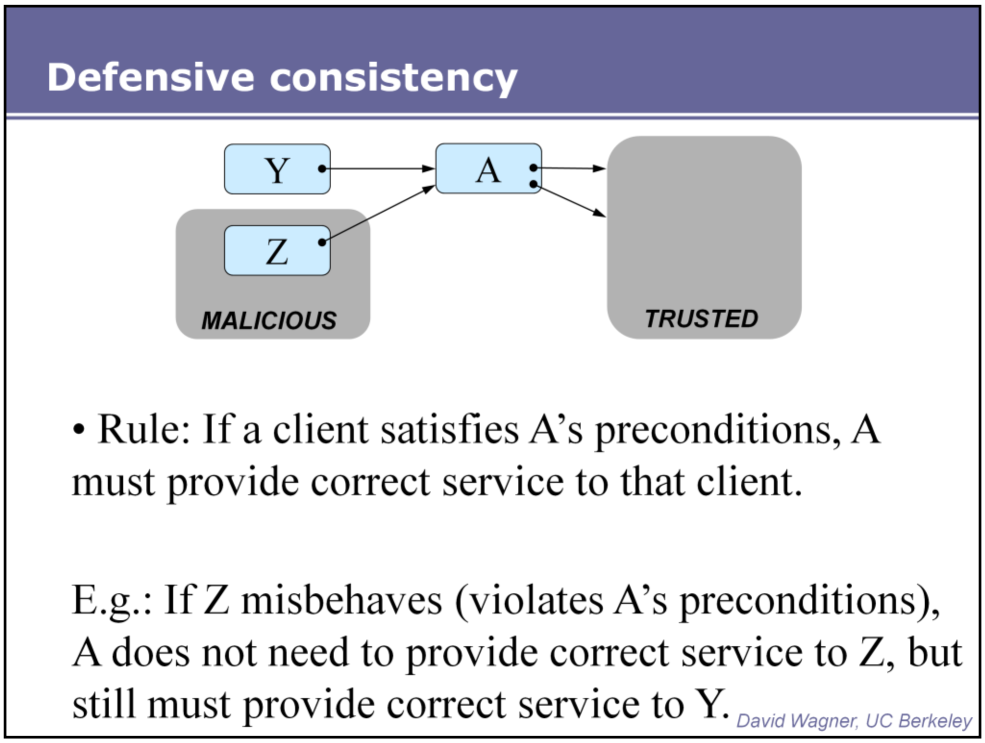
Alice is a client of Bob & Charlie; she relies upon them (is vulnerable to them).
Alice in turn provides services to her clients.

David Wagner, UC Berkeley

That brings me to the topic of trust, reliance, and vulnerability in object capability systems. In this picture, Alice provides service to her clients, Yolanda and Zack. At the same time, she might be a client of other objects, Bob and Charlie; she relies upon them to provide correct service to her and to abide by their contract with her. For instance, Bob and Charlie might be trusted class libraries provided by the system. This picture suggests that Alice trusts Bob and Charlie -- which is just a peculiar way of saying that she is vulnerable to their misbehavior. In particular, Alice may have decided to trust Bob and Charlie -- meaning she may have decided to accept being vulnerable to Bob and Charlie's misbehavior -- and we'd like our pattern of reasoning to be able to take that into account.

Fortunately, modular reasoning extends naturally to this case. We inspect the source code of Alice (which determines her behaviors), we examine the contract that Bob and Charlie promise to follow (which tells us what Alice can rely upon them to do), and we apply conservative reachability analysis to upper bound the possible behaviors of Alice's clients without examining their source code. If we can prove, from this information, that Alice upholds her end of her contract with her clients no matter how her clients behave, then we can mark Alice as trustworthy.

In this way you might think we could gradually build up a set of objects which have been verified to behave in a manner consistent with their contract. For instance, after verifying Alice, we might proceed to verify Alice's clients, and so on and so forth.



Defensive consistency

Y
A
Z
MALICIOUS
TRUSTED

• Rule: If a client satisfies A's preconditions, A must provide correct service to that client.

E.g.: If Z misbehaves (violates A's preconditions), A does not need to provide correct service to Z, but still must provide correct service to Y.

David Wagner, UC Berkeley

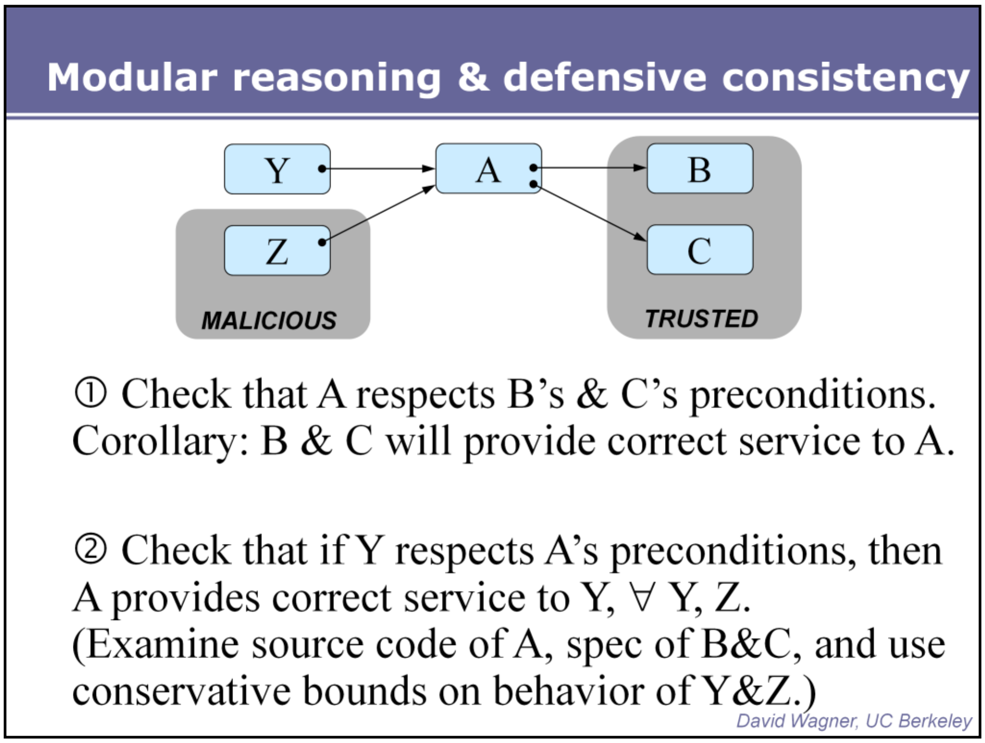
But that story about how to do modular reasoning is still too naïve. As stated, it requires Alice to provide correct service even if her client misbehaves. But that's unreasonable. If Alice can detect that her client is behaving maliciously, there is no reason we should hold her to the contract.

More precisely, a contract between Yolanda and Alice will typically specify some preconditions -- some obligations that Yolanda must meet -- and will promise to provide correct service only if those preconditions are met. The rule we want, then, is that if Yolanda is honest or otherwise establishes the necessary preconditions, then Alice must deliver correct service to Yolanda -- but if Yolanda is malicious and deviates from the contract by failing to live up to her obligations, then we release Alice from all obligations and make no demands upon how Alice behaves in that case.

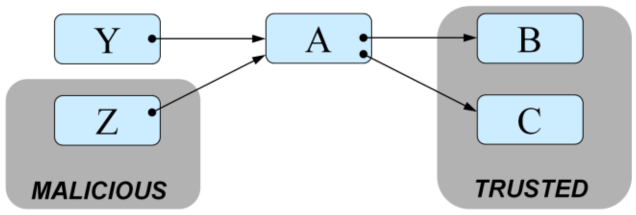
As discussed so far, this is the standard precondition/postcondition Eiffel-style programming with contracts. However, in object capability systems, a second issue pops up: Alice might have many clients, and some of those might behave correctly, while others might misbehave. In that case, we would usually like to contain the damage -- we'd like to prevent Alice's malicious clients from harming Alice's honest clients.

For instance, maybe Alice provides a MP3 encoding service. If Yolanda is an honest client of Alice, and Zack a malicious client, we don't want Zack to be able to attack Yolanda by confusing Alice into providing incorrect results to Yolanda.

"Defensive consistency" is a style of modular reasoning intended to deal with the many-clients issue. It says that Alice is obligated to provide correct service to clients that meet her preconditions, but she is released from all obligations to clients that misbehave.



Modular reasoning & defensive consistency



① Check that A respects B's & C's preconditions.
Corollary: B & C will provide correct service to A.

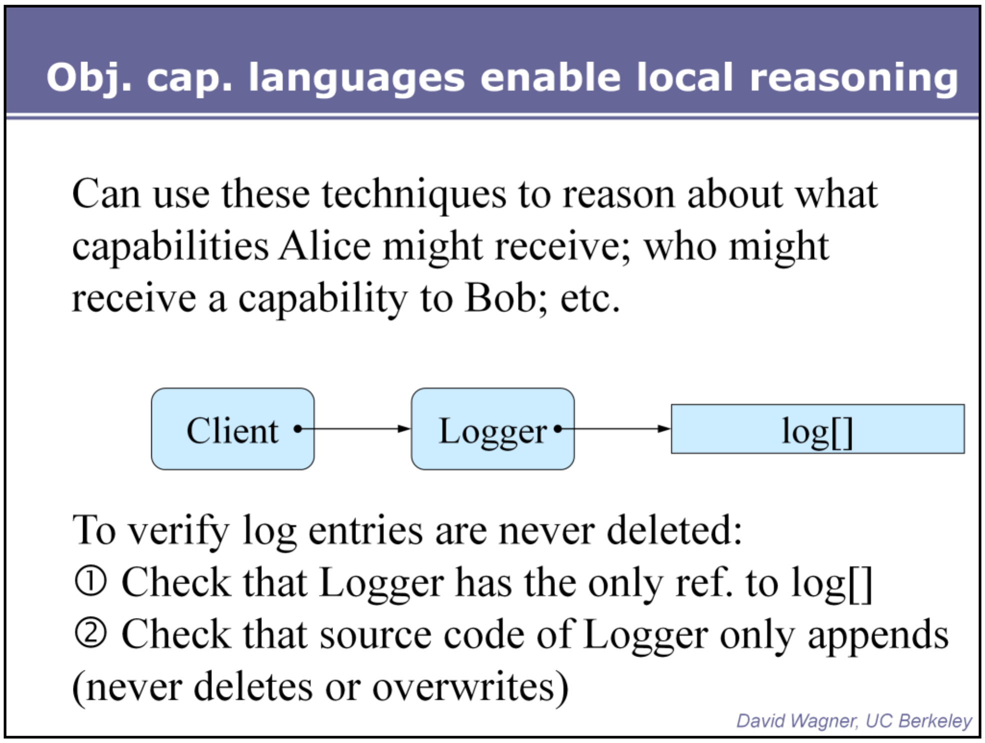
② Check that if Y respects A's preconditions, then A provides correct service to Y, ∀ Y, Z.
(Examine source code of A, spec of B&C, and use conservative bounds on behavior of Y&Z.)

David Wagner, UC Berkeley

Let's see how to verify defensive consistency with a more specific example. Suppose Alice relies upon Bob and Charlie to do some work for her. Then, to check that Alice is defensively consistent, we begin by checking that Bob and Charlie are defensively consistent and that Alice always establishes Bob and Charlie's preconditions any time she invokes them. Once we've done that, we're entitled to conclude that Bob and Charlie will provide correct service to Alice (and this is true no matter what how Bob and Charlie's other clients may behave! -- a crucial point). At this point, Alice can safely rely upon Bob and Charlie to provide correct service to her.

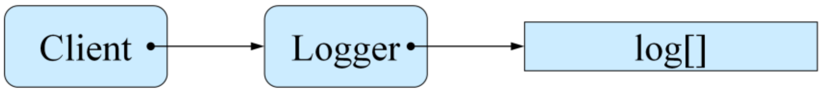
Next, we check that, for all possible ways to instantiate Yolanda and Zack, if Yolanda respects Alice's preconditions, then Alice will uphold her contract with Yolanda and provide correct service to Yolanda. We have to check that this is true no matter how Zack (and any number of other malicious clients of Alice) misbehave, which typically involves some kind of conservative reachability analysis applied to Zack. Finally, once all of these conditions are met, we are entitled to conclude that Alice is defensively consistent.

The attraction of defensive consistency is that it's a powerful way to compose larger systems out of smaller subsystems, because it lets you apply modular reasoning even in an open-world setting where some of the code might be malicious.



Obj. cap. languages enable local reasoning

Can use these techniques to reason about what capabilities Alice might receive; who might receive a capability to Bob; etc.



To verify log entries are never deleted:
① Check that Logger has the only ref. to log[]
② Check that source code of Logger only appends (never deletes or overwrites)

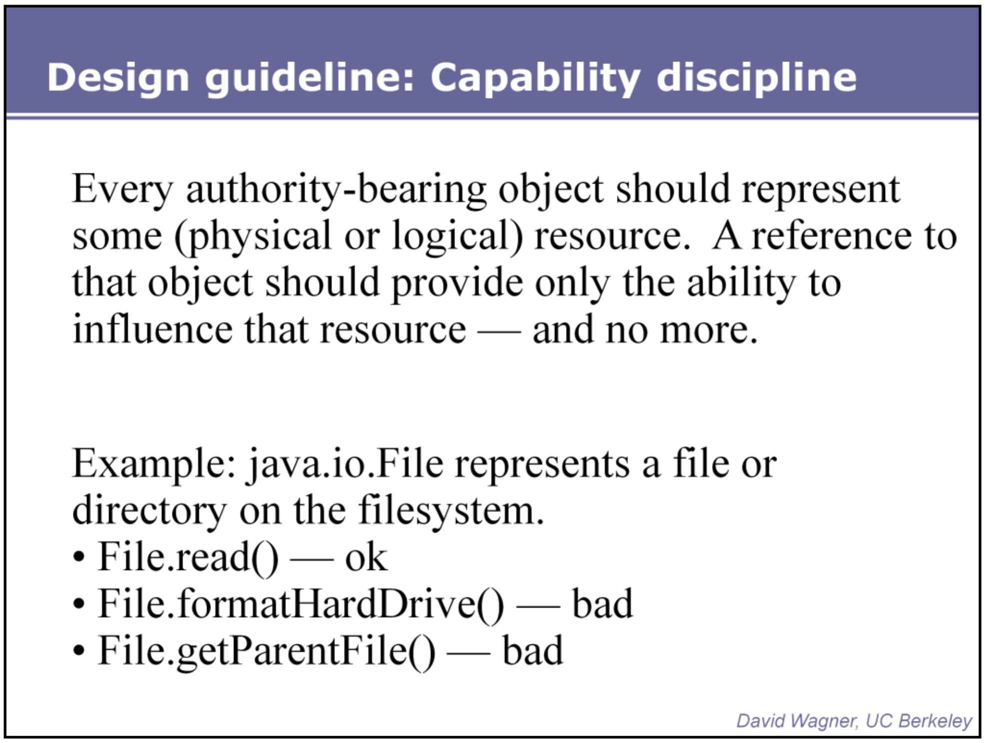
David Wagner, UC Berkeley

I'll recap where we've gotten by giving a simple example of how one might apply capability-style reasoning to build secure systems. Here we want to maintain an in-memory audit log that is supposed to be append-only, so that new log entries can be added but previously written log entries cannot be modified or deleted. The implementation illustrated here uses a component, the Logger, which receives a message to be logged and writes that message to the end of the log data structure. Suppose that this is written in an object-capability safe programming language. What do we have to do to verify that this log indeed follows an append-only discipline?

First, we identify all objects that might be able to obtain a capability to the log data structure. If we're lucky, the Logger was written to construct this data structure anew and is careful to keep the capability to it closely held, making sure that capability does not leak to anyone else. If the Logger was implemented thoughtfully, that property should be trivial to verify by inspecting the source code of the Logger. At this point we are entitled to conclude that the Logger is the only one with the ability to directly modify the contents of the log data structure. Next, we inspect the source code of the Logger to make sure that it can only perform append operations, and that it will never destroy or overwrite previously written log entries. That, too, should be easy to verify if we've implemented the Logger in a reasonable way. That's it -- you're done! At this point you can conclude that the log is truly append-only. Notice how easy that was?

In comparison, if we had written this as a monolithic application in C or C++, there would be no hope of verifying this property (short of tediously inspecting every line of source code in the entire application). In C or C++, there is no way to control who can modify the log data structure. Note that object capability languages impose restrictions on what kinds of programs you can express, but it's exactly these restrictions that enable us to reason in these powerful ways.

Of course, in real programs things are rarely this simple, but the advantages of capabilities at simplifying reasoning about security still remain significant, in my experience, and this is a pretty representative example of why.



Design guideline: Capability discipline

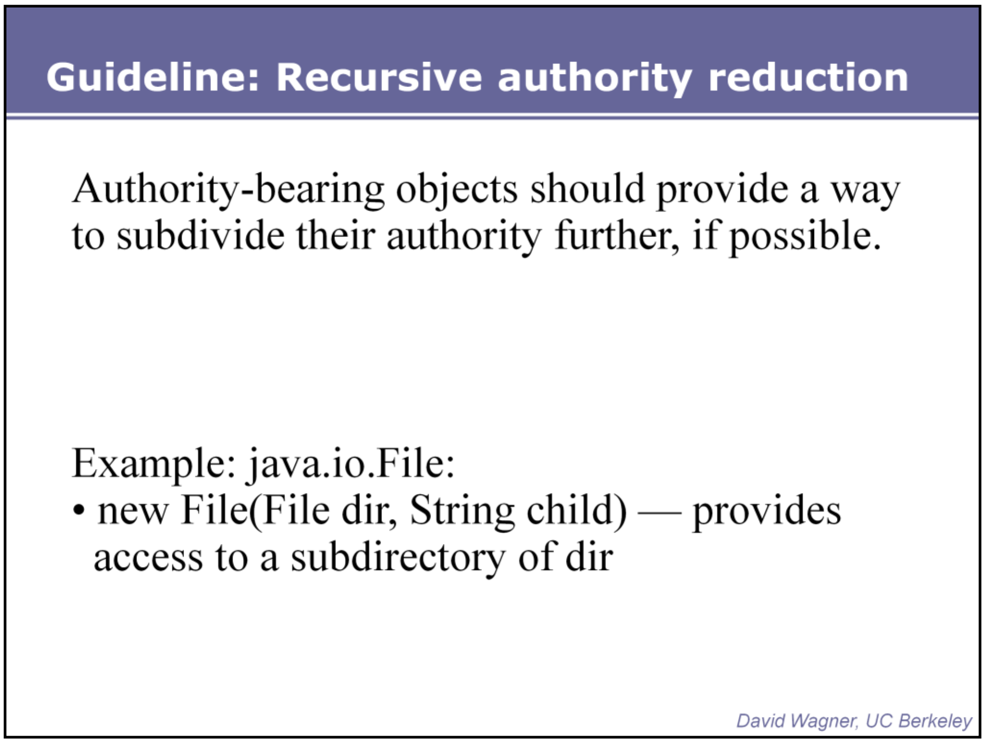
Every authority-bearing object should represent some (physical or logical) resource. A reference to that object should provide only the ability to influence that resource — and no more.

Example: java.io.File represents a file or directory on the filesystem.
- File.read() — ok
- File.formatHardDrive() — bad
- File.getParentFile() — bad

David Wagner, UC Berkeley

That concludes my discussion of capability-style reasoning. I now want to provide some hints for how to structure your program to make the most of the object capability approach. The first guideline, which goes by the name “capability discipline”, is about how to structure your class hierarchy and method interfaces. The idea is that there should be some clear story about what authority each object in the program represents. Moreover, the decomposition into objects should reflect the application’s security goals. For instance, if the desired security policy says that faculty should be treated differently than students, I might have a class People with two separate classes Faculty and Student to reflect this distinction.

This is probably easiest to explain with an example. Java’s class java.io.File is a familiar example that comes pretty close to following capability discipline. Presumably, the intent is that a java.io.File object should act as the representative of some file or directory stored on the filesystem. If so, capability discipline says that if I give you a reference to a java.io.File object, you should receive the authority to modify that particular file, and nothing else. Let’s see what implications this has for what kinds of methods a java.io.File object ought to support. A read() or write() method that reads or writes bytes from the file under consideration is fine, because they only provide authority to access the contents of that file. However, if the java.io.File class had a formatHardDrive() method, that would be a violation of capability discipline: if I gave you a reference to a java.io.File object, you would receive not only the ability to modify that file on the filesystem, but also the ability to format the entire hard drive, which exceeds the intended authority of java.io.File. Ok, that was a hypothetical example, but here’s a real one: java.io.File has an instance method getParentFile() that returns a java.io.File object representing the parent directory, I.e., the directory that this file lives in. That’s a violation of capability discipline, because it represents excess authority. For instance, if I gave you a reference to a java.io.File reference, you could iteratively call getParentFile() until you had a reference to a java.io.File object representing the root directory -- and that gives you authority to trash the entire filesystem. Not good. If java.io.File had been designed according to capability discipline, it would not have had a getParentFile() method.



Guideline: Recursive authority reduction

Authority-bearing objects should provide a way to subdivide their authority further, if possible.

Example: java.io.File:

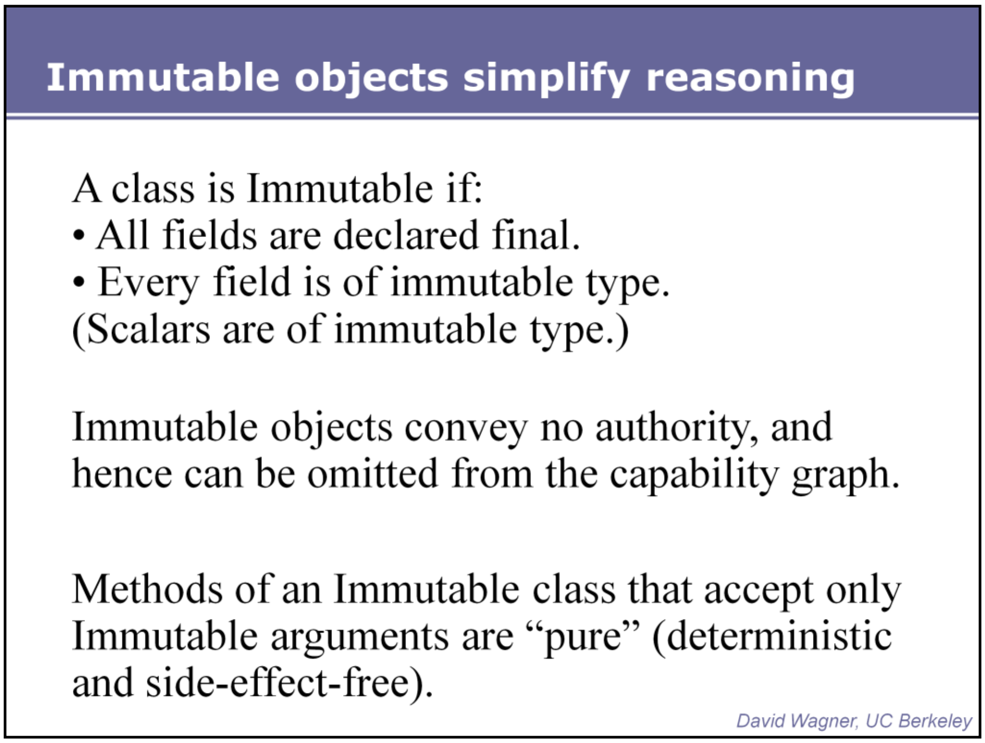
- new File(File dir, String child) — provides access to a subdirectory of dir

David Wagner, UC Berkeley

A second design guideline: If I have a capability to some coarse-grained object, it should be possible to carve that up into smaller pieces and give other people capabilities to just the pieces they need.

For instance, suppose I have a capability for the root directory on the filesystem. That capability represents the authority to write to any file on the filesystem. In Java, given a capability for some directory D, I can derive a capability for any subdirectory of D. That derived capability will provide authority to modify only a subtree of the filesystem directory structure, not the entire filesystem. It is possible to continue subdividing authority like this into smaller and smaller bits, until the result is as small as needed.

This principle is useful, because it allows code to subdivide its authority and hand out just those pieces that are needed to services it invokes. This enables you to do a better job of respecting POLA, because it gives you a way to create your own abstractions of authority at as fine a granularity as you desire.



Immutable objects simplify reasoning

A class is Immutable if:
• All fields are declared final.
• Every field is of immutable type.
(Scalars are of immutable type.)

Immutable objects convey no authority, and hence can be omitted from the capability graph.

Methods of an Immutable class that accept only Immutable arguments are "pure" (deterministic and side-effect-free).

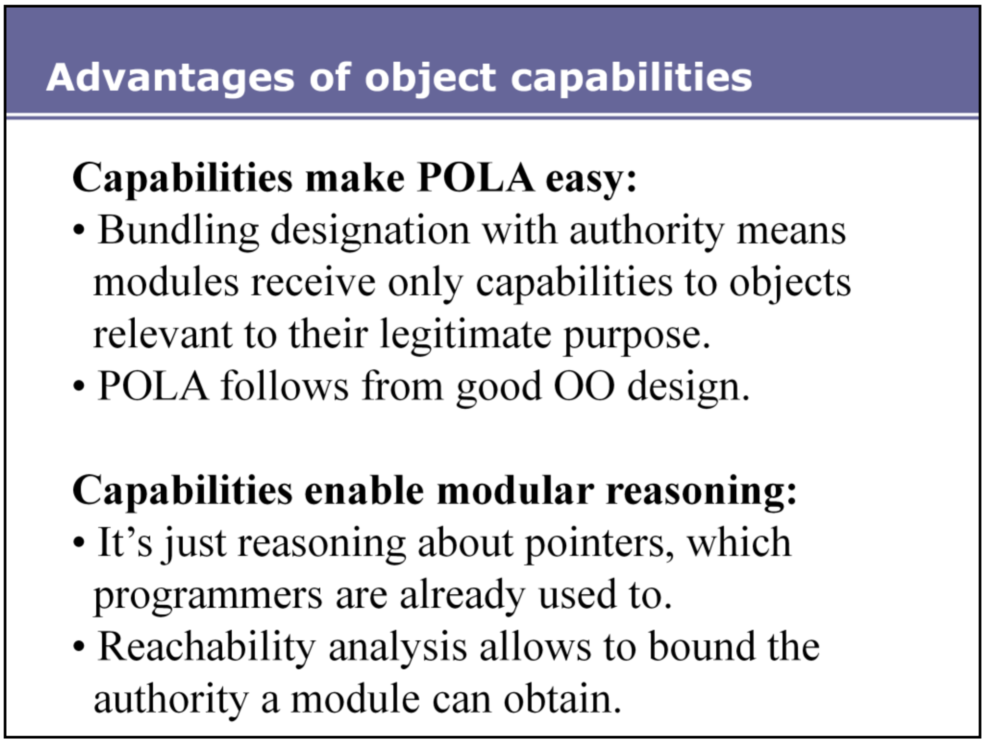
David Wagner, UC Berkeley

Another observation is that immutable objects often simplify capability-style reasoning. The reason is that mutable objects create authority -- namely, a reference to a mutable object conveys the authority to modify the contents of that object -- while immutable objects convey no authority. The integer 2 doesn't convey any authority; I can give you a copy of the integer 2, and it doesn't increase your authority. The same is true of any immutable type.

Because immutable objects convey no authority, they can be omitted from the capability graph. This makes it easier to do capability-style reasoning, because the graph just got smaller. If you and I share a reference to an immutable value, that shared reference doesn't create any overt communication channels that would let us pass capabilities between us; it can be safely ignored for the purposes of conservative reachability analysis.

A second very cool property of immutable objects is that they provide "purity theorems for free". If I have an immutable class with a method that accepts only immutable arguments, then in a capability-based language, I am guaranteed that the method will be observationally pure: it is side-effect-free and its return value is some deterministic function of its arguments.

Thus, a useful pattern in capability systems is to use immutable objects where possible.



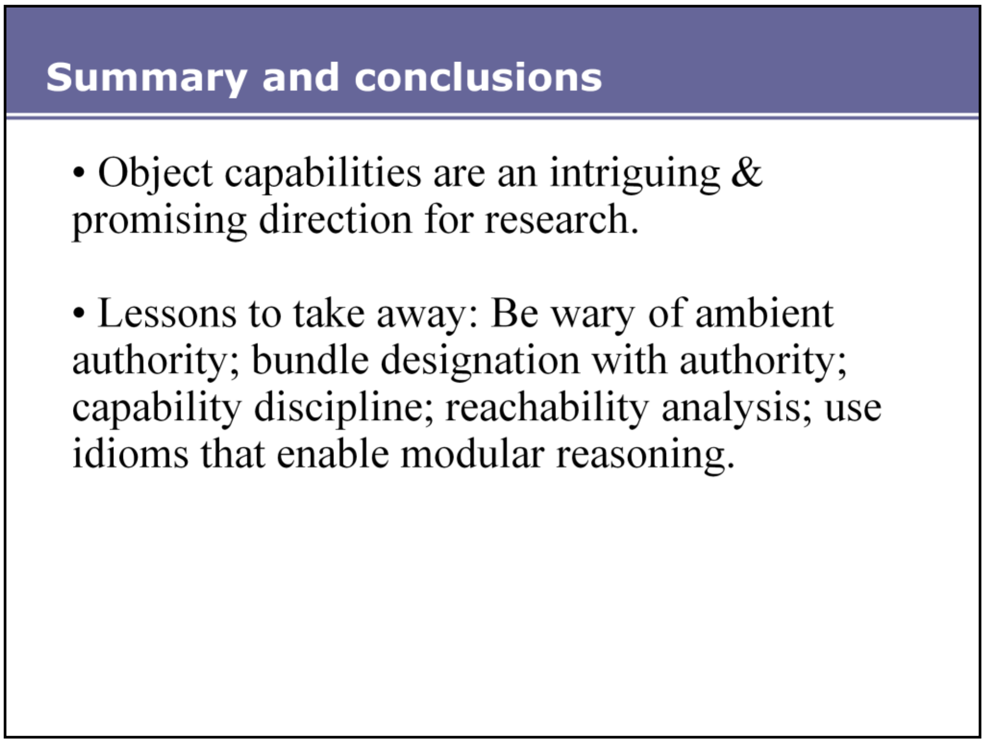
Advantages of object capabilities

Capabilities make POLA easy:
- Bundling designation with authority means modules receive only capabilities to objects relevant to their legitimate purpose.
- POLA follows from good OO design.

Capabilities enable modular reasoning:
- It's just reasoning about pointers, which programmers are already used to.
- Reachability analysis allows to bound the authority a module can obtain.

So, to summarize the contents of this talk, object capability systems have two intriguing advantages: first, they make POLA come almost for free, because references do double duty of both designating which things you want to operate on and simultaneously providing the authority needed to do so; and second, they enable modular reasoning about authority.

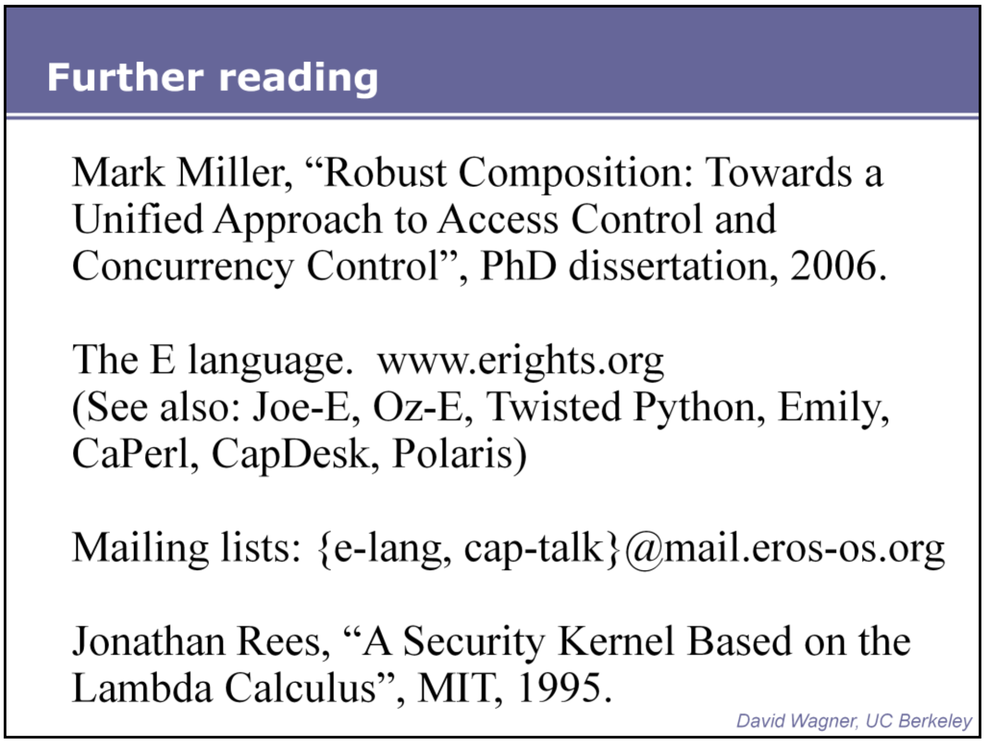
If you're familiar with privilege separation, you can think of the object capability philosophy as enabling "privilege separation on steroids", where the app is decomposed into many mutually distrusting subsystems, each following POLA. The difference is that privilege separation has traditionally been applied at a process granularity, which makes it such a pain to build security perimeters that developers rarely bother, or at best divide their apps into two or three subsystems, and that limits how much progress one can make towards achieving POLA. In contrast, object capability systems are intended to make it really easy to erect security perimeters, so easy that you can create lots of them and do a much better job of practicing POLA.



Summary and conclusions

- Object capabilities are an intriguing & promising direction for research.
- Lessons to take away: Be wary of ambient authority; bundle designation with authority; capability discipline; reachability analysis; use idioms that enable modular reasoning.

In summary, I think the capability community has come up with some pretty intriguing ideas. Now you've seen enough that you can judge for yourself. And you can put them into practice the next time you're building a security-critical system: you can look for opportunities to minimize ambient authority, to bundle designation with authority, to choose your class hierarchy in a way that respects capability discipline, to design your code to be defensively consistent, and to practice using simple reachability analysis to reason about the flow of authority in your system.



Further reading

Mark Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control", PhD dissertation, 2006.

The E language. www.erights.org
(See also: Joe-E, Oz-E, Twisted Python, Emily, CaPerl, CapDesk, Polaris)

Mailing lists: {e-lang, cap-talk}@mail.eros-os.org

Jonathan Rees, "A Security Kernel Based on the Lambda Calculus", MIT, 1995.

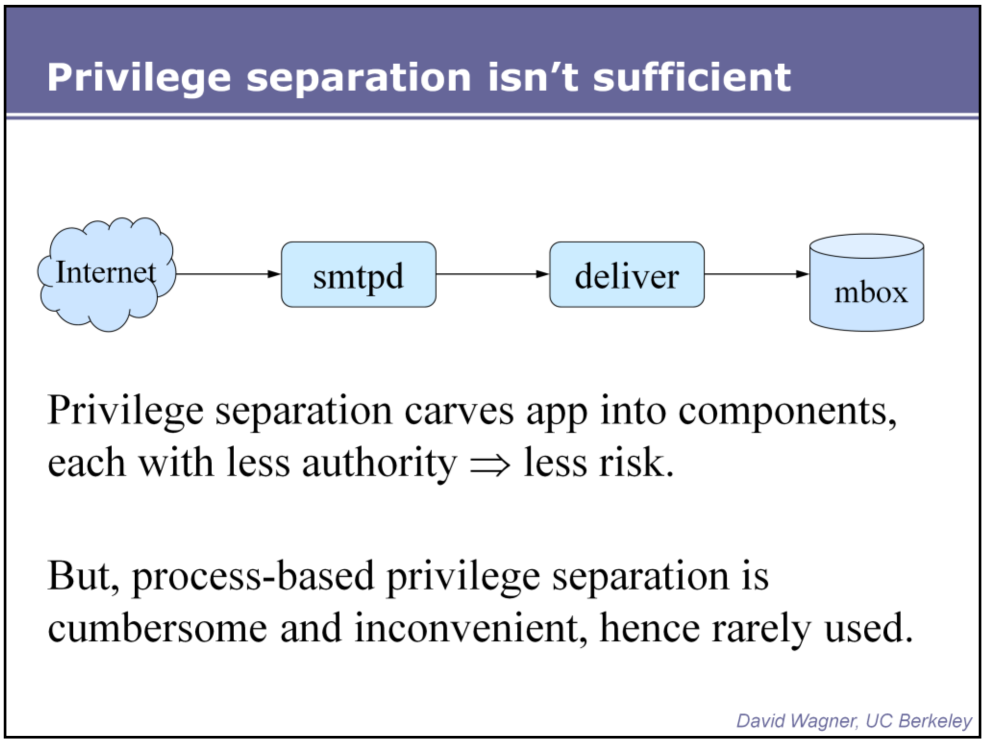
David Wagner, UC Berkeley

If you'd like to learn more, here are some references.

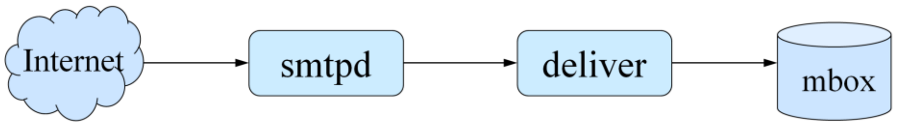
Thank you for your attention. I would welcome any questions or comments you might have.

Extras

David Wagner, UC Berkeley



Privilege separation isn't sufficient



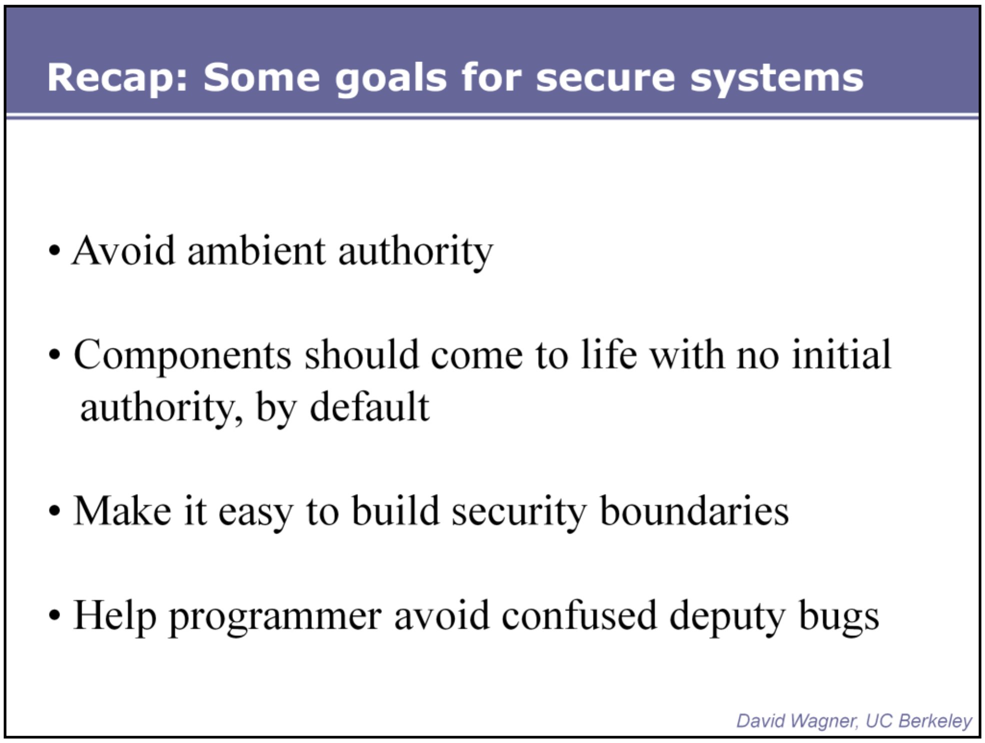
Privilege separation carves app into components, each with less authority $\Rightarrow$ less risk.

But, process-based privilege separation is cumbersome and inconvenient, hence rarely used.

David Wagner, UC Berkeley

One approach people have studied is to use privilege separation: to break the app up into smaller pieces, chosen so that each piece needs much less authority. For instance, you can imagine a sendmail-replacement with one process that listens on port 25 and speaks SMTP, and another process that accepts an email and delivers it to the right mailbox file on the filesystem. This reduces the harm from security compromises, because the SMTP daemon doesn't need any access to the filesystem, and the delivery agent doesn't need access to the network. However, today privilege separation is crude and cumbersome. Operating systems don't provide good support for privilege separation, and consequently it is highly inconvenient. (For instance, you spend a lot of effort marshalling up data structures and passing copies to other processes, since you don't have a shared heap.) Because it is such a pain to introduce security boundaries into the system, developers do so only very sparingly and reluctantly. Typically, privilege separation is not used at all, or is used at only a coarse granularity. It would be better to have a system that makes it easier to introduce security boundaries -- lots of them.

A second problem of privilege separation is that each process still uses ambient authority. As you start carving the system up into more and more mutually distrusting components that need to interact with each other, you run into another serious pitfall of ambient authority: it encourages confused deputy problems.



Recap: Some goals for secure systems

- Avoid ambient authority
- Components should come to life with no initial authority, by default
- Make it easy to build security boundaries
- Help programmer avoid confused deputy bugs

David Wagner, UC Berkeley