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

Lecture 15: Capabilities

Lecturer: Matt Fredrikson

Note: Portions of the material in this lecture were adapted from notes by David Wagner.

15.1 Confused Deputies, Authority, and Least Privelege

An illustrative story. We begin today's class with an old story due to Norm Hardy, dating back to the 1970s. I've updated the details of this story so it uses syntax and commands that are more modern, but the main idea is the same. Hardy worked on a time-sharing machine that was probably decommissioned before most of us in this class were born. The compiler on this system maintained a charge log that kept track of billing information about everyone who used it, so that they could be charged for their use of the limited computing resources. So, when the user called the compiler as follows, it would append a line to `/var/log/charges` noting the use.

```
$ cc prog.c -o prog
```

At one point, one of the users observed that if you fed the compiler the name of the charge log as its output filename, then the compiler would overwrite the billing file with the compiled output of their source.

```
$ cc prog.c -o prog
$ cc foo.c -o /var/log/charges
```

A simple hack to obtain free compilation! Let's think about what went wrong here, coming from the perspective of what authority the compiler needs to do its job on this machine.

- The compiler needs to be able to access all of the files under the invoking user's control, both to read the source file and write the executable.
- To support the billing scheme, the compiler also needs write access to `/var/log/charges`.

Obviously, we don't want `/var/log/charges` to be world-writeable, or even writeable by most users. It suffices to ensure that *only* the compiler has access to this file. In effect, we "deputize" `cc` with the task of mediating access to `/var/log/charges`, trusting it to do the right thing. However, we found a way to trick `cc` into abusing its authority, creating what is called a **confused deputy** that abused its authority on our behalf.

Ambient authority. To see the underlying problem, we'll introduce some vocabulary that's conventional in the capabilities literature. The first term we need is **authority**, which we've used before in the context of access control and authorization logic. Its meaning here is similar but perhaps less precise, as we'll take it to mean *the set of ways in which a program can cause changes to the external system state*. Examples of authority would be means of reading and writing files, changing pixels on the screen, reading and writing the network, and things of that sort.

Authority refers to the set of ways that a program can cause changes to external system state.

It's important to note that authority is transitive. If Alice has the authority to modify `/var/log/charges`, and Bob has the authority to talk to Alice, then there may be some message that Bob can send to Alice that causes her to modify `/var/log/charges`. Importantly, this might be true even if Bob doesn't have direct authority to modify `/var/log/charges`. Unless we have good reason to believe otherwise, the conservative thing to do is assume that Bob's ability to talk to Alice transitively grants him all the authority of Alice.

The second term we'll use is **ambient authority**, whose definition is similarly imprecise. Ambient authority is the authority posessed by a program at any particular time, that it did not necessarily request or need. Think of ambient authority as the authority that's "floating in the air", available to the program if it wants it.

Ambient authority is the authority posessed by a program that it did not necessarily ask for, and may not necessarily need.

An example might help us gain some intuition about ambient authority. This example is from David Wagner. Consider two ways of copying a file on Unix systems. The first, and probably the one that most of us would think to use involves `cp`.

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

What authority does `cp` need to allow us to use it this way, for whatever files we'd like? Generally speaking, it needs broad authority to read and write any files that we as the user also have access to. When we run `cp` on a particular pair of filenames, it inherits all our powers on the filesystem, even though it only *needs* authority on the single set of filenames we passed it. These unnecessary priveleges are ambient authority available to `cp`.

An alternative way of copying files is to use `cat`. The `cat` utility is pretty flexible, so there are a number of ways we could do this, but one of them involves only passing the utility pipes to files that we already opened.

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

When we use `cat` in this way, it doesn't need any authority of its own when it comes to our files. The piping operators cause the shell to open `foo.txt` for reading and `bar.txt` for writing, and simply pass the corresponding file descriptors to `cat`. Even though `cat` isn't written this way in current systems, it could be written so that it has no authority other than that passed by the user each time it is invoked. If this were the case, `cat` would have no ambient authority.

Least privelege. We've talked about least privelege several times before, in general as an objective that programmers should always strive to achieve. You might have guessed that all this time we've been talking about ambient authority, we've really been talking about least privelege. Ambient authority is harmful specifically because it violates least privelege — unnecessary, unrequested authority is as clear-cut an example of such a violation as we'll see.

This is what allowed us to create a confused deputy in the `cc` example. The compiler is given ambient authority to read and write all our files, and also to modify the charge log in `/var/log/charges`. No restrictions were placed on how the compiler was allowed to change the charge log, so we were able to trick it into producing changes that should not have been possible.

15.2 Capability systems

Let's go back to the compiler example, and take a closer look at what transpired. Supposing the compiler were written in OCaml (unlikely, as OCaml did not exist in the 1970s), its simplified main body might look something like the following.

```
let main (s: string) (d: string) =
  let src = read_contents s in
  let obj = compile src in
  let _ = write_charge "/var/log/charges" in
  write_contents d obj
```

When this code is invoked, it will run with the ambient authority of the user, in addition to the right to modify the charge log. Whoever wrote the compiler probably expected that the latter authority would only be used in the fourth line, in the call to `write_charge`. However, nothing about the way this program is written enforces that expectation, and the operating system will only check that `cc` has write priveleges on `/var/log/charges`. This is why we were able to create a confused deputy out of this code.

How can we fix this code to un-confuse the deputy? Let's think back to the example of `cp` vs `cat`. Much like `cp`, the compiler is currently written to accept strings representing filenames that it needs ambient authority to access. Perhaps we can rewrite the compiler to behave more like our idealized version of `cat`.

```
let charge_fd = open_out "/var/log/charges"

let main (s: in_channel) (d: out_channel) =
  let src = read_contents s in
  let obj = compile src in
  let _ = write_charge charge_fd in
  write_contents d obj
```

In this version of the code, the caller is expected to open the input file `s` and output file `d`, and subsequently pass them to the compiler. Because the user should not have the authority to open `/var/log/charges` for writing, there is no way to trick the compiler into calling `write_contents` on the charge log.

Importantly, notice that this version of the code does not need the ambient authority that was necessary to support opening files from arbitrary filename strings passed in by the user. We can configure the system so that the compiler only has the authority to open `/var/log/charges` for writing, and nothing else. This design embodies least privelege, and is inherently resistent to confused deputy attacks like the one we've been discussing.

Capabilities. The solution given above uses an abstraction called **capabilities**. Capabilities are an abstraction that simplifies the practice of implementing least privelege. The central idea is to bundle **designation** and **authority** together in a single object that can be used by programs. By designation, we mean a unique name or logical address of a resource that the program operates on. As before, authority essentially means the set of priveleges conferred on the designated resource.

Capabilities are not an abstract concept, they correspond to tangible things in programs that use the capability model. In our running examples so far, file descriptors are capabilities. When a program is passed a file descriptor, it is given the authority to access the file associated by the operating system with that descriptor, in the manner consistent with the access priveleges with which the descriptor was created.

```
let fd = fopen "foo.txt" "r"
bar(fd)
```

In this program, the file descriptor `fd` is a capability. It confers **read** authority on the file designated with the name `foo.txt`.

Programs that adhere to the capability model do not require ambient authority—any change that they wish to effect on the rest of the system must be accompanied by a capability that is given to the program by someone with the appropriate authority. This leads to programs that need significantly less authority than they would had they not been written using capabilities, and simplifies the task of specifying access control policies. The person who invokes a program in the capability model implicitly specifies what authority that program should receive, and can do so in a discretionary fashion.

Object capabilities. There are many ways to implement capability systems, but today we'll focus on one particular paradigm called object capabilities. In an object capability system, the universe consists of **objects** and **capabilities**.

Objects correspond to anything that that might convey authority (i.e., affect the rest of the system)—functions, programs, files, network, hardware, and any resources that we might want to subject to access control.

Capabilities are *unforgeable references* to objects. Capabilities can be held by other objects, and confer the ability to send messages between objects.

The message-passing capacity of capabilities is unidirectional. If A holds a capability to B, then A can send messages to B; B cannot send messages to A unless it holds the corresponding capability. In object capability systems, the only way that a program can make changes to the rest of the system is by sending messages along capabilities. Obviously, this means that in order to change an object, a program must hold a capability to that object in order to send a message along it. Thus, because capabilities are **unforgeable**, capabilities have a complete monopoly on authority in an object capability system: there is no way to gain authority outside of capabilities.

An important point is that when objects are created, they possess no authority. The only way for an object to gain authority is for another object to pass it a capability, so when an object is initialized it has only the authority given to it by its creator. Notice how different this is from the examples of ambient authority we discussed earlier—in an object capability system, least privelege is the default, and the only way to violate it requires active intervention by an object with authority.

Language support. In terms of primitives available in conventional programming languages, capabilities can be seen as pointers and objects as instances of type with protected state and well-defined entry points. This seems to suggest that many object-oriented languages, which typically provide ways of expressing both of these sorts of things, could be a good match for writing code that adheres to the object capability paradigm.

However, we need to be careful when attempting to use object capabilities in languages that were not explicitly designed to support them. Let's look at a few object-oriented languages, and think about whether they're appropriate.

C++: Being a descendent of C, C++ represents pointers as unsigned integers. Importantly, C++ gives the ability to cast unsigned integers as pointers, which then point to an arbitrarily-chosen location in memory. This is problematic for object capabilities, because casting allows one to create pointers out of then air, or in other words forge a reference. This is a fundamental problem, as it prohibits us from even creating a specialized `Capability` class with private state containing the reference: a programmer can create a pointer to exactly the location where the object reference is stored, thus gaining the authority granted by the capability.

Javascript: Although it's object oriented and supports unforgeable references, Javascript is a hot mess when it comes to thinking about object capabilities. This has been the topic of a good deal of research in the past, culminating Google's Caja system and a number of related language subsets put out by other companies and academics. We won't get into the details of what one needs to do to make Javascript suitable for object capabilities, but suffice it to say that this is a non-trivial undertaking.

Java: Java is a better candidate for implementing object capabilities, following from its memory safety. Java does not allow programmers to create pointers to arbitrary objects, they must either come from a call to `new` or be passed in from another method. The language's type-checking facilities support private state (instance variables) and well-defined entry points (methods) that cannot be tampered with as in C++. Thus, capabilities can be implemented with pointers to Java objects. However, a few features in Java violate the object capability paradigm, so object capability programs need to be written in a subset of Java (see Joe-E by Mettler and Wagner). For example, native methods might undermine type safety, static fields are globally-accessible and thus provide ambient authority, and reflection might allow programmers to forge references. However, it's important to note that the object capability model can be obtained by *removing*, rather than adding, features in the language.

OCaml: Like Java, OCaml is actually somewhat close to implementing what's needed for object capabilities. References are unforgeable, and its type system is more sophisticated than Java's. A few undocumented features like `Array.unsafe_set` could violate memory safety, so need to be disabled when using object capabilities. A number of pervasives confer ambient authority (such as `open_out`), static mutables must be disabled (similar to static fields in Java), and external calls to native code should be disallowed. As in the case of Java, removing language features suffices to provide a viable object capability language.

A number of real languages support object capabilities, and are used to gain assurances in a number of domains including web programming, smart contracts, and general system security. For more information on these languages, see the list contained in the Wikipedia page for the object capability model.

15.2.1 Reasoning about authority.

One of the key benefits of object capability systems is that they make it easier to reason about the flow of authority in a program. For example, suppose that we want to write a function that parses and displays image data which comes from an HTTP connection, and therefore cannot be trusted. We probably want to ensure that this function can't obtain any authority to access the filesystem, but we also want to make sure that it has the authority to update video memory. For another example, suppose we want to write a function that updates the contents of `/etc/shadow` when a user changes her password. We need to reason about what other parts of the code might be able to access the authority that this function has.

Object capabilities support this type of reasoning by ensuring that authority must follow capabilities, i.e., object references. To determine what authority our image-parsing function has, we only need to determine the set of references passed to it by other functions. This amounts to reasoning about the flow of pointers in a type-safe language, which is a skill already familiar to many programmers.

Reference graphs. We'll use graphs representing object references when we reason about authority in the object capability model. Each object in the system is a node, and a directed edge from A to B denotes a capability (i.e., a reference) to B held by A.

$$\begin{tikzcd} A \arrow[r] & B \end{tikzcd}$$

We can associate such a graph to the state of a program at a given point in time. As the program executes, the graph will change. There are three basic transformations that we need to reason about how authority can flow as the program executes.

1. First, we assume that any object can obtain a reference to itself. This is modeled by allowing the graph to change at any time by adding self-loops on arbitrary nodes, with a transformation rule called Self.

$$\begin{tikzcd} A \arrow[r, Rightarrow, "\textsc{Self}"] & A \arrow[loop right] \end{tikzcd}$$

2. The second rule deals with object creation. We assume that whenever an object creates a new object, it can obtain a reference to it. The transformation rule is called, unsurprisingly, Create.

$$\begin{tikzcd} A \arrow[r, Rightarrow, "\textsc{Create}"] & A \arrow[r] & B \end{tikzcd}$$

3. The third rule is more interesting, and deals with delegation. If A has references to B and C, and A wishes to share her reference to C with B, then we assume she may do so. This is captured by the transformation rule Intro.

$$\begin{tikzcd} & B & & & & B \arrow[dd] \\ A \arrow[ur] \arrow[dr] & & & & A \arrow[ur] \arrow[dr] & \\ & C & {} \arrow[r, Rightarrow, "\textsc{Intro}"] & {} & & C \end{tikzcd}$$

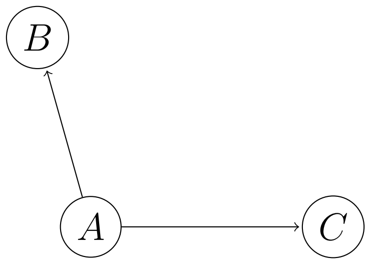
Confinement. By repeated application of these rules, we can reason not only about what authority an object might have at any given moment, but also what authority it might obtain in the future by interacting with other objects. In particular, if we want a conservative over-approximation of the authority that an object might obtain throughout the execution of the system, then starting from an initial configuration we can proceed to transform the graph by applying each of these rules whenever it is applicable. This leads to the following theorem.

Theorem 15.1 (Conservative bound on authority) *Let* $\mathbf{R}$ *be the edge relation corresponding to the initial state of a program's reference graph, and let* $\mathbf{R}^*$ *be the reflexive, symmetric, transitive closure of* R*. If* A *and* B *are objects in* R*, then* A *can influence* B *if and only if* $A\ \mathbf{R}^*\ B$*.*

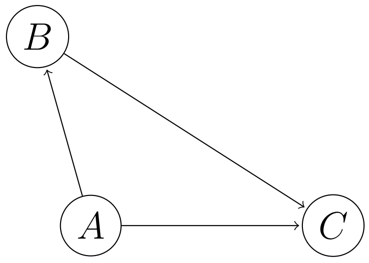
This theorem tells us that if A and B are totally disconnected in the initial reference graph, then A will never have authority on B throughout the lifetime of the system. Importantly, this tells us that object capability systems allow us to enforce policies that establish **confinement**, an important and cross-cutting property that is useful in many settings.

Revocation. One often-cited drawback of systems that model capabilities as unforgeable references is that they do not support revocation of authority, or the ability to take away authority that was previously granted. **Is this true?**

To get to the bottom of this, let's look at a scenario involving revocation. Suppose that A creates an object C, and wishes to give an existing object B, for which A already has a capability, authority on C that can later be revoked.



The most direct way to grant B authority to C is for A to pass the reference directly, corresponding to an application of Intro.

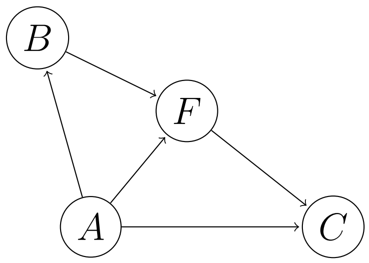


What does A do when it wants to revoke B's access? We're in a tight spot, because nothing we've discussed allows us to do away with edges. Perhaps we can assume another transformation, Kill, which says that if A creates C, then A can remove C entirely from the graph (perhaps corresponding to calling `delete` on the appropriate reference). This certainly seems like a viable option, but now A has also lost authority on C because it no longer exists in the system.

A different solution uses indirection and composition to achieve revocation. A creates a *forwarding* object F, performing the following actions.

1. A passes F a reference to C.
2. A passes B a reference to F.

This leaves us with the following graph.



The forwarding object F does what its name suggests: any message it receives from another object is forwarded along the only capability it has, to C. However, it also listens for special messages from its creator A, which tell it to drop its capability on C.

Notice that to model this behavior, we need to give objects the ability to drop references. This is reasonable in object capability systems, as we can imagine setting a pointer to `null`. We can model this as a transformation rule `Drop`, which says that outgoing edges can be removed from any node.

$$\begin{tikzcd} A \arrow[r] & B \arrow[r, Rightarrow, "\textsc{Drop}"] & A & B \end{tikzcd}$$

Thus, when A invokes its capability on F, the graph corresponding to our scenario will look like the following.

$$\begin{tikzcd} B \arrow[dr] & & \\ & F & \\ A \arrow[uu] \arrow[ur] \arrow[rr] & & C \end{tikzcd}$$

Now, A has not explicitly revoked anything from B, as A did not create B and thus cannot assume any control over its state. B still maintains a reference to F, but once A invokes F this reference is useless because it can no longer mediate access to C.

Let's see what this would look like in code, using OCaml as our language. We'll start by defining the code for F, as this is where the interesting parts are. To keep things relatively general and simple, we'll assume that all objects define `send` method, along which messages for capabilities are communicated.

```
class F c_in = object
  val mutable c_ref = Some c_in

  method send msg =
    match c_ref with
    | Some c -> c#send(msg)
    | None -> ()

  method revoke = c_ref <- None
end;;
```

We implement a class that takes a single argument, a pointer to an object C, for initialization. F keeps its reference to C in an option type, so that on initialization it keeps the value `Some c`, which is a reference to C, but later on can be changed to drop this reference by storing the value `None` instead. The `send` method does what we discussed earlier: it forwards all messages it receives to C's `send` function. However, it only does so if the current value of `c_ref` contains a reference to some object. Then we can define A as follows.

```
class A b_in = object
  val b = b_in

  method main =
    let c = new C in
    let f = new F c in
    // send B the revokeable capability on C
    let _ = b#send f
    // some additional code ...
    // before exiting, revoke B's capability on C
    f#revoke

  method send msg = ...
end;;
```

We won't bother with implementing B or C, because their details aren't important. What is worth pointing out is that B can call `send` on the instantiation of F that it is passed from A, just as it would if A would have instead passed a direct reference to C. It does not need to know about the internal details of the reference it receives, and has no way of accessing them to violate the encapsulation we're trying to achieve (assuming we're working in an appropriate subset of OCaml).

Modular reasoning. When we talked about confinement, we discussed how to over-approximate the authority that objects in a program might accrue over time by taking the reflexive, symmetric, transitive closure of the intial reference graph. In many cases, this approximation might be too coarse, and lead us to believe that objects we're concerned about will receive too much authority. It might be safe to assume that all possible graph transformations will be applied, but if we write our program appropriately, we may be able to, for example, assume that A will never share a reference to C with any other object.

By analyzing the code implementing our system, we can in some cases accomplish this. However, in order for this to scale we want to restrict ourselves to **modular** reasoning, where we can examine each function or component of the implementation in isolation, making minimal assumptions about the rest. Object capabilities allow us to do this reasonably well.

The general pattern that we would like to follow is illustrated in the picture below. Suppose that we would like to reason about the authority of A, who can be accessed by Y and Z, and who has capabilities on B and C.

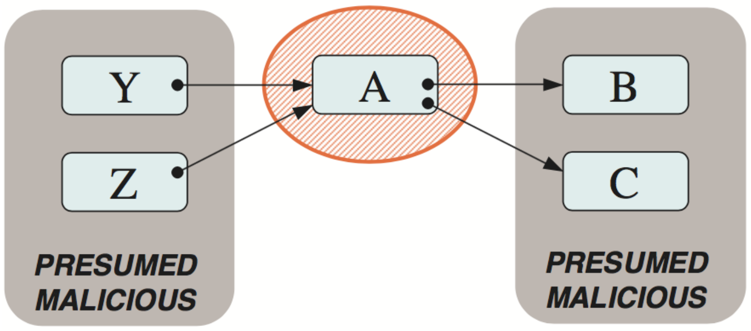


Image credit: David Wagner

We are willing to analyze A's code directly, but we cannot look at the code that implements the other objects in any greater detail than what is needed to reconstruct the reference graph. In fact, we might as well just assume that the reference graph has been given to us directly as input, and that we can't observe the implementation of B, C, Y, Z at all.

Our goal is to prove things about what will happen within A regardless of what happens among the objects that we presume to be malicious. The only assumption that we make is that they are also written with object capabilities, so we can upper-bound the authority of these objects using the reflexive, transitive, symmetric closure result discussed previously. We then combine this with the facts that we have inferred about A's behavior to draw a final set of conclusions about the authority of A. Notice that depending on how well we did analyzing A, this could lead to more precise results than if we had applied the conservative reachability analysis to the entire system. However, the results will still be sound: we essentially quantified over all possible implementations of B, C, Y, Z in the object capability model.

Defensive consistency. Now let's assume that A has reason to trust all of the objects for which it holds capabilities. This is a common requirement when reasoning about realistic systems, as it is difficult to get anything done if we aren't allowed to trust the operating system, standard libraries, and similar services in our system. Now our situation looks like the following picture.

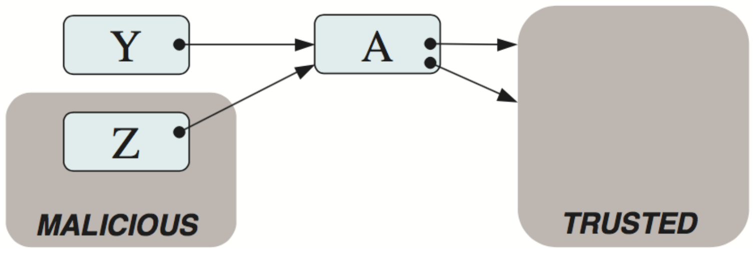


Image credit: David Wagner

We'll assume that contracts exist between A and its trusted objects, as well as between Y, Z and A. The type of contract that we're interested in here is similar to what you'll recall in C0, where functions are always annotated with pre and post-conditions so that whenever a function's preconditions hold when it is called, it is guaranteed that its postconditions will hold when it returns.

Modular reasoning extends naturally to this case, and should hopefully allow us to make stronger conclusions and claims about the behavior and authority of A. However, in object capability systems, we're interested making a particular kind of consistent guarantee about the behavior of our objects. Specifically, A might have many clients, some of them malicious and others honest.

As with normal contractual guarantees, A is only obliged to fulfill her postcondition if the preconditions are met, so in this example if Z violates a precondition then A does not need to provide anything in return. However, we would also like to guarantee that regardless of what Z does, A is always able to fulfill its contract with Y whenever the preconditions are satisfied. Implementations that satisfy this property are said to have **defensive consistency**.

Modular reasoning can help us verify defensive consistency in implementations that use object capabilities.

1. First, we either verify or assume that all of A's trusted objects satisfy defensive consistency.
2. Then, we need to ensure that A will always receive correct service from its trusted modules, call them B and C. To do so, we must verify that any time A invokes a capability for a trusted module, it satisfies the preconditions and will thus enjoy the postconditions.

3. Next, we need to show that no matter how Y and Z are instantiated, as long as Y satisfies A's preconditions, then A will satisfy its postconditions for Y. Importantly, because Z may not satisfy its preconditions, we need to make conservative assumptions about its authority using the conservative reachability analysis previously discussed.

4. Once we've verified that these conditions are met, we can conclude that A is defensively consistent.

This is an effective way to compose larger systems from smaller components. Modular reasoning and defensive consistency allow us to make strong conclusions even in an open-world setting where some objects might be malicious.

Programming guidelines. There are a few guidelines that programmers can follow when writing code in the object capability model that will help them achieve the potential security properties that it offers.

Capability discipline: This guideline posits that every capability should represent a resource, either physical or logical. Any reference to that object should confer only the ability to influence that resource, and nothing more.

A reasonable example of this, due to David Wagner, is Java's `java.io.File` class. One would hope that `java.io.File` objects represent a specific file or directory present on the local filesystem. By capability discipline, this implies that a reference to such an object should only give us the ability to access that particular file or directory, and nothing else on the filesystem (or elsewhere). Reasonable methods that follow this discipline might be `read` or `write`, as these only refer to the contents of the file. An unreasonable method might be `formatHDD`, as this confers the ability to modify not just the file designated by the object, but the entire filesystem at once. Essentially, such a method would provide a substantial amount of ambient authority to objects that have `java.io.File` objects.

`java.io.File` comes close to adhering to capability discipline, but it does indeed violate it in at least one way. The method `getParent` returns an reference to an object that designates the parent directory. This is an example of ambient authority in the API, as we can probably assume that not every caller of `java.io.File` has reason or need for a reference to the parent directory.

Recursive authority reduction: This guideline posits that capabilities should provide a way to subdivide the authority they grant, whenever possible. In other words, if I have a coarse grained capability, perhaps `java.io.Filesystem`, then it should be possible to chop it up into smaller pieces, i.e. `java.io.File` objects, to pass to other objects. When combined with the capability discipline, this guideline supports implementing least privelege with ease; programmers won't be tempted to pass sweeping coarse-grained capabilities to other objects because they can easily prune down such authority to create new capabilities, and the capabilities thus created will only authorize changes to the relevant constituent resources.

A real example of this is embodied in the `java.io.File` constructor `new File(File dir, String child)`, which allows one to create a new capability with authority over some child directory from a capability on its parent.

Immutability: Whenever possible, using immutable objects simplifies reasoning about object capabilities. The reason is simple: immutable objects confers no authority to change its contents. They are effectively constant values, and being given a constant value does not grant any authority. Immutable objects can be removed from the reference graph, thus simplifying the reasoning we discussed earlier and in some cases leading to more precise results.