Access Control for IoT: A Position Paper

Alan H. Karp

Introduction

We get work done with our computers almost in spite of their usability, rather than because of it. This statement is particularly true when it comes to collaboration, even when "collaborating" with ourselves across our own machines. We manage to cope on our computers and smart phones. We won't be able to when IoT is ubiquitous.

Marc Stiegler has identified the root cause of the problem; our devices do not support aspects of sharing that we rely on in the physical world. These aspects can be illustrated with two stories.

> *In an emergency, Marc asked me to park his car in my garage. I couldn't do it, so I asked my neighbor to do it for me and told her to get the garage key from my son.*

I doubt that anyone would think twice about this story. The second story is in the computer domain.

> *In an emergency, Marc asked me to copy a file from his computer to mine. I couldn't do it, so I asked my neighbor to do it for me and told her to get access to my computer from my son.*

People often find this second story so preposterous that they laugh.

In what follows, I'll describe the six aspects of sharing and score some existing sharing tools against them. Next, I'll show why our commonly used access control mechanism cannot (NB: not ***do not***, but ***cannot***) support the kinds of sharing that we need. I'll then describe an approach to access control that naturally supports the six aspects of sharing and illustrate it with examples from our work. Finally, I'll propose a way to apply this approach to IoT devices and propose some standards we need to develop.

The Six Aspect of Sharing

Figure 1 illustrates the six aspects of sharing.

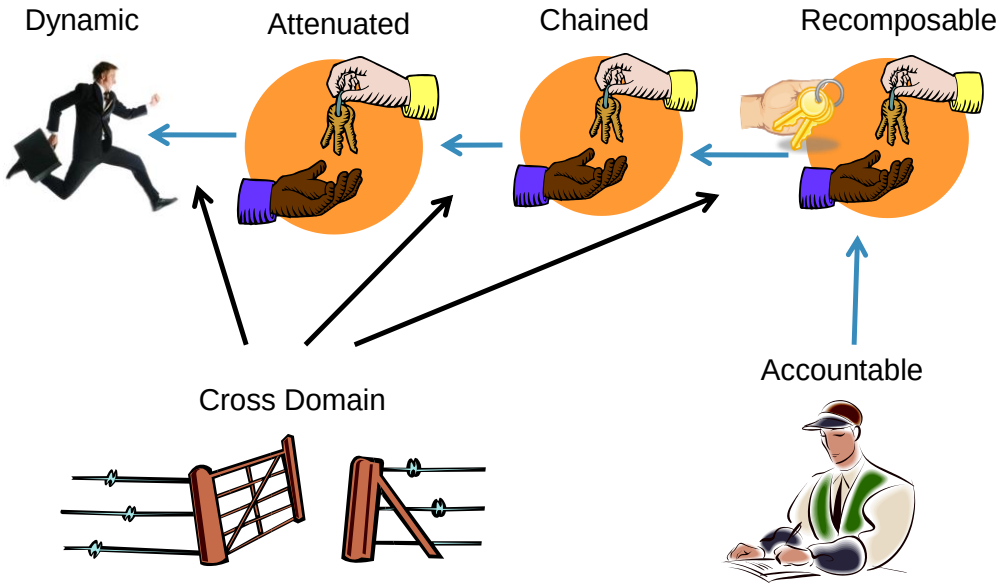

Figure 1: The six aspect of sharing that we rely on in the physical world.

1. Dynamic: It's an emergency, so there's no time to find an administrator to change permissions.
2. Attenuated: Marc did not have to give me all his permissions, *e.g.*, his login password; he can just give me his car keys.
3. Chained: I could pass Marc's car keys to my neighbor. I could even further attenuate by giving her just the valet key.
4. Composable: My neighbor combines one permission she got from me, the car key, with one she got from my son, the garage key, to complete the task.
5. Cross-jurisdiction: There are three families involved, each with its own policies, yet there's no need to communicate policies to another jurisdiction. In the example, I didn't need to ask Marc to change his policy to grant my neighbor permission to drive his car.
6. Accountable: If Marc finds a new scratch on his car, he knows to ask me to pay for the repair. It's up to me to collect from my neighbor. The systems we use today would show my neighbor was responsible but wouldn't record the fact that I'm the one who asked the administrator to give her permission.

Some of these aspects are absolutely critical. Without chaining, every Army private is saying, "Yes, sir, Mr. President." Without attenuation, that private ends up with permission to launch nukes.

A common complaint is, "But haven't you lost control by allowing all this delegation?" The fact is that any control that you might think you had is an illusion. People will find workarounds, and those workarounds usually violate the assumptions behind your security architecture, resulting in worse security. For example, it is common for people to share credentials if that's the only way they can get their work done. By blocking the delegation of a little bit of permission, you force people to share all of theirs while losing accountability.

Table 1 shows how well a few tools support the six aspects of sharing. SharePoint makes sharing particularly hard, but even innovative tools like Google Wave come up short. Only sharing by exchanging email attachments meets all the requirements. Unfortunately, that's copying, not sharing. Even if we could do true sharing with email, that's hardly an acceptable approach for IoT.

Table 1: Scoring sharing tools on the six aspects of sharing.

	SharePoint	LiveMesh	Wave	Email
Dynamic				
Attenuated		Only on the first hop		
Chained		But not attenuated		
Composable				
Cross domain	Setup dependent			
Accountable				

Access Control

Access control is the way we enforce a key element of the policy needed to secure our devices. A common way to conceptualize the access policy is with the *access matrix*, an example of which is shown in Table 2. Each row corresponds to a *subject*; each column, to an *object*. The cells of the table specify the permissions the subject in the row has to the object in the corresponding column.

Table 2: A sample access matrix.

	File1	File2	File3
Alice	R		R,W
Bob		R	R
Carol	R,W	R,W	

Since an actual access matrix is far sparser than in this simple example, it's more efficient to store it in compact form. A record for each object of which subjects have which permissions is a data structure called an Access Control List (ACL). In this example, the ACL for File 1 is (Alice[R],Carol[R,W]). Taking the other cut gives us a Capability List (CL). For example, the CL for Alice is (File1[R],File3[R,W]).

It would seem that there's no particular reason to pick one cut over the other, but there is. Consider the simple case of the UNIX function cp. Carol wishes to copy the contents of File1 to File2, so she issues the command **cp File1 File2**. The question is how the process running the cp command gets permission to read File1 and write File2. Since there's no convenient way in an ACL system to grant the process those specific permissions, the cp process must run with all of Carol's permissions, which has some unfortunate consequences. For one, Carol can't specify which of her permissions goes with which argument. Should she mistakenly issue the command **cp File2 File1**, the contents of File1 will be lost.

More importantly, it means that every process Carol runs must run with all her permissions, which is why malware is such a problem. [1]

Things just work with capabilities. Instead of specifying strings to denote the two files, Carol passes open file handles, *e.g.*, **cp <File1 >File2**. The process running cp gets exactly the permissions needed to carry out Carol's request and no more. Passing capabilities as arguments has the effect of updating the access matrix exactly when and as necessary. The upshot is that the damage malware running in the process running cp can do is limited to File2. Note that if Carol specifies the arguments in the wrong order, the request fails because the process will try to write to a file opened without write permission.

Why How We Do Access Control Today Cannot Work

ACLs have their origins in the time sharing systems of the 1960s. They've been stretched, twisted, and crumpled to make them work as we moved from mainframes to client server to personal computing to the Internet to mobile. It's been a *tour de force* by some very smart people. However, the mismatch between the original environment and where ACLs are being used today leads to many of the problems we're forced to deal with every day. IoT will only exacerbate those problems.

The critical element in what we do today is that the requester's authentication is used to make the access decision. This choice has unfortunate implications for sharing. Marc must update the policy on his computer to give me permission to access the file he wants me to copy. I can't access the file until I set up an account on his machine and figure out how to authenticate to it. Worse yet, I have no way to create an account on Marc's machine for my neighbor. The only practical way for Marc's file to get copied is if I give my neighbor my credentials to Marc's machine.

Additional problems arise in other situations we're likely to encounter with services designed for IoT. It's quite likely that an IoT device will connect to a cloud service for storage and computation. Problems arise when that cloud service invokes a third party service to satisfy the request from the device.

Say that I use a cloud-base voice recognition service to control both my television and my thermostat. When I say something to the voice recognition service via my TV app, the TV app authenticates to the voice recognition service as me. That service must authenticate to the recommendation service, but what authentication does it use?

If the voice recognition service uses its own credentials, I could have constructed a request for something the recommendation service has permission to do but I do not. Perhaps the result of my request will tell me about my neighbor's viewing habits. We say the voice recognition system has become a confused deputy [2].

If the voice recognition service uses my authentication when invoking the recommendation service, it now has permission to do anything I have permission to do, whether or not I want it done. There might be unfortunate consequences if my request to my TV app was "Fahrenheit 451," but the voice recognition service uses my authentication when mistakenly forwarding the request to my thermostat.

I have seen no solution to these problems that isn't equivalent in its essential features to what I propose in the next section. Roles don't address the fundamental problem, nor does using attributes. The root cause of the problem is that the authentication presented with the request is necessarily independent of the request, which means the authentication necessarily represents all the permissions assigned to me.

There are no such problems with capabilities. My TV app delegates to the voice recognition service my capability to the recommendation service. Since that capability carries only my permissions, a request for my neighbor's viewing habits will be rejected. Similarly, the request will be rejected by my thermostat if the voice recognition service makes that mistake.

authoriZation-Based Access Control (ZBAC)

A capability is an unforgeable token that both designates a resource and grants permission to access it. On a single machine, a capability could be an entry in hardware managed by the operating system on behalf of a process [3]. In a pure object oriented language, such as the subset of Java that passes the Joe-E verifier [4], an object reference is a capability.

Things get more complicated when we extend capabilities across machines, as we need to do to make full use of IoT devices. A URL with an unguessable query string or fragment has the property we need to use it as a capability. However, current browsers and web servers were not designed to protect secrets in URLs. That's why OAuth 2 puts the secret in the header with a tag denoting that it is a bearer token. Since the designation of the object, the URL, is separated from the designation of permissions, the *access token*, an OAuth 2 bearer token is not a capability. That doesn't mean we can't use the combination as a capability; it's just that extra care is needed. I introduced the term *authoriZation-Based Access Control (ZBAC)* to denote both such systems and capabilities [5].

Understanding how to use ZBAC starts by examining the access control process, which consists of four parts.

1. Identification: Knowing whom to hold responsible for authorized actions. This step is often done in person for banking or employment; certificate authorities provide this function for online interactions.
2. Authentication: What allows a process use permissions assigned to an identity. Users authenticate to a computer, which in turn starts a process that has the ability to prove that the process is acting as an agent of the authenticating user.
3. Authorization: Granting a permission, which is the way we express policy. It associates an authentication (Credential sharing means not an identity.) with one or more permissions.
4. Access decision: Deciding whether or not to honor a request. This decision most commonly depends only on authorizations created before the request is made, but it can also depend on context. For example, an authorization can be contingent on some external property, such as time of day, or an event, such as the USA going to war with Canada.

We are all familiar with these steps. Identification is looking at my co-worker standing next to me when creating a new account on my server. In Microsoft Windows each process is given an *access token (*a misnomer as it's really an authentication token) that proves to the operating system that the process is allowed to use any of the permissions granted to the user who started the process. Putting an entry in ACL is an act of authorization. Finally, the invoked object or a process acting on its behalf must decide if the request should be honored or rejected.

Once we have identified these four steps, we can ask when each should be done, and where if the user and service are in different jurisdictions. Our options for each step are before the request is made or at request time and in the requester's domain or that of the service. These options are shown in Table 2.

In autheNtication (Identity, Role, Attribute)-Based Access Control (NBAC), the last three steps are done in the service domain. In ZBAC, the first three are done in the user domain before the request is made.

Table 3: Access control options.

	IBAC/RBAC/ABAC		ZBAC	
	Where	When	Where	When
Identification	User domain	Before request	User domain	Before request
Authentication	Service domain	At request time	User domain	Before request
Authorization	Service domain	Before request	User domain	Before request
Access Decision	Service domain	At request time	Service domain	At request time

The key feature of ZBAC is that the authorization decision is made in the requester's domain. It is this feature that simplifies permission management when collaborating across jurisdictions. After all, if my company signs a contract with your company, you have no way of knowing which of the permissions granted to my company should be granted to me; only my company knows. With NBAC, that policy information needs to be communicated. With ZBAC, my company gets a ZBAC token from your company and delegates to me a ZBAC token authorizing the subset of the permissions it wants me to have. There is no need to communicate policy to another jurisdiction, or for me to authenticate to any other jurisdiction, both of which can be important for maintaining privacy. Hence, there's no need for single sign-on or to federate identities. A more comprehensive comparison has been done [6].

Applying ZBAC to IoT

A common concern is how people are going to manage all these fine-grained permissions. As shown by CapDesk [7], we can use acts of designation in the UI, such as drag and drop, as acts of authorization. For example, dragging the icon for a file onto the icon for a word processor tells the system to start a process to run the word processor and grant it permission to read and write the designated file. This insight gives us hope that average users won't even know that they're managing permissions.

A simple example illustrates an interaction design that might be applied to IoT. Say that I get a flat tire. Today, I call AAA and wait with my car until the truck arrives. With IoT, I can use my AAA app to call for help from the nearest Starbucks. The app puts an icon representing the service call into a myIoT app's UI as shown in Figure 2.

I can click on the icon for my car key, which pops up a tab for each permission, and drag the icon for opening the trunk to the AAA icon. Similarly, I can click on the icon for my phone and drag the Text icon to the AAA icon. Under the covers, this process transfers the designated capabilities to AAA, and AAA can delegate those capabilities to whichever service company they choose. That company can in turn delegate the trunk capability to the person they send to change my tire. When I get a text notifying me that my car is ready, I can drag the AAA icon off the desktop, which results in revoking the granted capabilities.

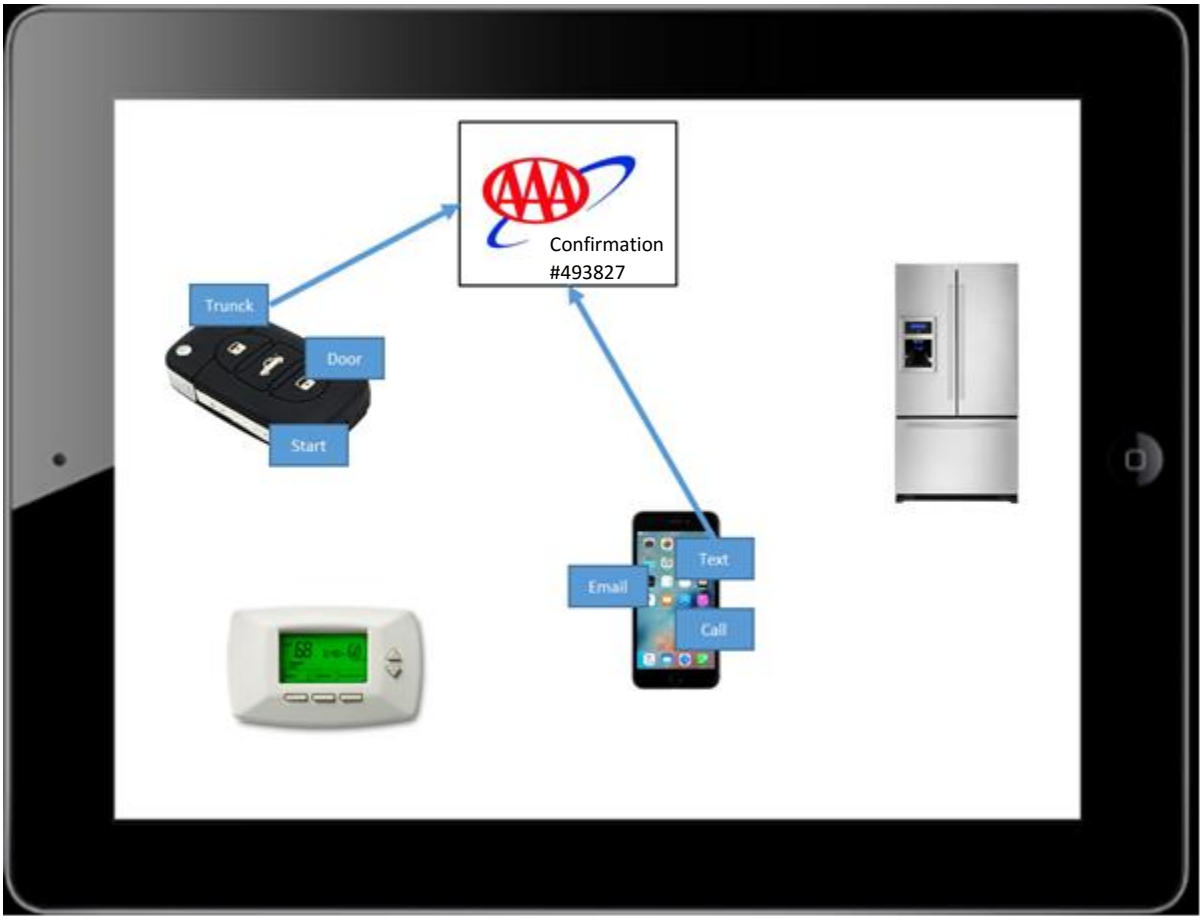

Figure 2: A possible interaction design for managing IoT permissions.

Notice that there is nothing that appears to be security-related to the user. That's not a unique feature of this use case; we've done it before. We were so successful in hiding security in our Secure Cooperative File Sharing (SCoopFS, the "F" is silent) tool that one of our users asked us how to turn on security [8].

Recommendations

ZBAC is a better match for the way we use our computers today than are the authentication-related mechanisms we currently use. I hope I've convinced you that it will be even more important as we move into the world of IoT.

I've only briefly mentioned the representation of the capabilities. E-speak capabilities were Simple Public Key Infrastructure (SPKI) certificates [9], and our Zebra Copy paper shows how to use SAML assertions as ZBAC tokens [5]. This approach is likely to be too heavyweight for many IoT devices.

Unguessable URLs are a good choice for REST capabilities, but there are risks due to the way URLs are handled by web servers and browsers. OAuth 2 bearer credentials probably make better ZBAC tokens

for IoT devices and services that use HTTP. One problem with these choices is that there is no standard for chained, attenuated delegation, which is an opportunity for an IEEE standards group.

The digital certificate approaches, SPKI and SAML, allow us to create attenuated delegation certificates locally. There are proposals for doing attenuated delegation of unguessable URLs and OAuth tokens, but they require a round trip to a third party. Macaroons [10] use hashes as capabilities, which are lighter weight than certificates while allowing local construction of attenuated capabilities.

In addition to a standard representation of capabilities, we need an agreed upon API for managing them if we hope to implement an interaction design like the one described in the previous section. We must start on these standards before the IoT world becomes embedded in our lives. If we don't, we'll end up with walled gardens, an AOL of Things, if you will.

References[i]

1. Polaris: Virus Safe Computing for Windows XP, M. Stiegler, K.-P. Yee, T. Close, A. Karp and M. Miller, Communications of the ACM, vol. 49, #9, pp. 83-88, September (2006).
2. The Confused Deputy: (or why capabilities might have been invented), N. Hardy, ACM SIGOPS Operating Systems Review, Volume 22, Issue 4 (October 1988).
3. Programming semantics for multiprogrammed computations, J. B. Dennis and E. C. Van Horn, *Commun. ACM* 9, 3 (March 1966.
4. Joe-E: A Security-Oriented Subset of Java, A. Mettler, D. Wagner, and T. Close. NDSS 2010.
5. Access Control for the Services Oriented Architecture, A. Karp and J. Li, ACM Workshop on Secure Web Services, ACM #459074, pp. 9-17, Fairfax, VA, November 2007
6. From ABAC to ZBAC: The Evolution of Access Control Models, Journal of Information Warfare, vol. 9, #2, pp. 37-45,September 2010.
7. E and CapDesk: POLA for the Distributed Desktop, M. Stiegler and M. Miller, http://www.combex.com/tech/edesk.html.
8. Making Policy Decisions Disappear into the User's Workflow, ACM Conference on Human Factors in Computing Systems (CHI 2010), Work-In-Progress, April 10-15, Atlanta, GA.
9. E-speak E-xplained, A. Karp, CACM, vol. 46. #7, pp. 113-118, July (2003)
10. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud, A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable and M. Lentczner, Network and Distributed System Security Symposium, 2014

[i] Self-plagiarism alert: I have copied material from my previous publications and presentations without attribution.