# Access Control (v0.1)

Ben Laurie
(benl@google.com)

## 1. INTRODUCTION

Access control is central to computer security. Traditionally, we wish to restrict the user to exactly what he should be able to do, no more and no less.

You might think that this only applies to legitimate users: where do attackers fit into this worldview? Of course, an attacker is a user whose access should be limited just like any other. Increasingly, of course, computers expose services that are available to anyone – in other words, anyone can be a a legitimate user.

As well as users there are also programs we would like to control. For example, the program that keeps the clock correctly set on my machine should be allowed to set the clock and talk to other time-keeping programs on the Internet, and probably nothing else[1].

Increasingly we are moving towards an environment where users choose what is installed on their machines, where their trust in what is installed is highly variable[2] and where "installation" of software is an increasingly fluid concept, particularly in the context of the Web, where merely viewing a page can cause code to run.

In this paper I explore an alternative to the traditional mechanisms of roles and access control lists. Although I focus on the use case of web pages, mashups and gadgets, the technology is applicable to all access control.

## 2. ACCESS CONTROLS BASED ON ROLES

Of course, in reality, we are never actually controlling the user's access – instead we are controlling programs that act on his behalf. Traditionally we have taken the view that the program itself is trustworthy: that is, that it will faithfully carry out the wishes of the user who is controlling it. For that reason, traditional access control has focused on controls based on the identity of the user running the program.

In practice we use roles rather than users because of messiness in the real world:

- Sometimes there isn't really a particular person associated with an activity we'd like to control: for example, if we are running a nuclear power plant, the entities we wish to be in control are identified by their role, such as "engineer in charge" or "janitor". It is the role we wish to grant permission to, not the person.
- People come and go. When somebody starts a job they acquire a new role (or roles), and when they leave they lose the role.
- People are fallible: it is useful for their own safety to limit their powers to their current role so that mistakes are less costly.

But increasingly we are moving to a world where the user cannot trust the program he is running. Even if we look at traditional environments the devolution of control away from the data centre and professional IT staff and towards the end user himself means there are programs running on peoples' machines that they have no idea how much to trust, and nor do they have any way to evaluate those programs.

If we look at the Web, particular the modern trend towards mashups, gadgets and single-domain[3] applications (e.g. Twitter, Flickr, Dopplr) we see this problem in spades. Applications are webpages, users switch between applications, including to completely new ones, at the click of a mouse. With mashups and gadgets the user is not even running a single program but multiple programs from multiple sources all sharing the same page.

Yet we are still trying to control everything with access controls based on roles (ACBR)[4].

[1] Perhaps it should also be allowed a little long-term storage, for example to keep its calculation of the drift of the native clock.

[2] A user probably trusts their operating system more than their browser, their browser more than the pages they browse to and some pages more than others.

[3] Single problem domain, that is, not DNS domain.

[4] I avoid saying "role-based access controls" because that has a rather specific meaning.

This makes no sense at all. The user has no way to sensibly create roles and the permissions associated with them. Even if they had, the task of assigning those roles to the various components of, say, a web page would be impossible.

Of course, I would not be saying all this if I did not think there was a better way. There is: capabilities.

A capability can be thought of as a "handle" representing some operation on some object. Possession of the capability is all that is required to perform that operation. The security of a capability system then boils down to your ability to control who[5] has each capability.

Some examples of capabilities are:

- Read this particular file.
- Write the same file (note that this is a completely different capability from the read capability).
- Pay money into my bank account.
- Pay money into your bank account.
- Take money out of my bank account.

Each of these capabilities is completely independent of all the others. I cannot derive a read capability from a write capability, nor vice versa. I cannot take money out of your bank account just because I can pay it in.

Note, however, that it is possible to derive new capabilities from old ones, for example:

- Write only well-formed HTML to some file, derived from the capability to write to the file.
- Write a safe subset of HTML to some file (for example, banning `<script>` tags and other unsafe constructs, derived from the HTML capability above.
- Transfer money from my bank account to your bank account, derived from the capability to take money out of my bank account and the one to put money into yours.

Note that neither of these can do any more than the capabilities they are derived from (of course: how could they?) – indeed, in both cases they do less. In fact, this is generally likely to be the case for derived capabilities, otherwise why bother to derive them?

In order to turn capabilities into a security system, we need some extra properties...

- Capabilities are unforgeable – that is, the only way to possess a capability is to be given it, or to derive it from capabilities you already have.
- All access to resources (that is, anything outside the program itself) is through capabilities.

In practice, it is usually simplest to build capabilities on traditional object-oriented programming. In this approach, an object corresponds to a group of capabilities which operate on some underlying state (in the case of read access to a file, the object might let me do seek operations as well as read operations on the underlying file object), often itself represented as another object which might be shared between multiple capabilities (my bank account would be shared between the capability to deposit money in it and the capability to withdraw money from it). Deriving new capabilities from old then becomes business as usual – create a new object, include references to the relevant existing capabilities and away you go.

When capabilities are implemented in this way you need to impose a further property on the objects: encapsulation. That is, it should not be possible for the holder of a capability to look inside it and see its inner state[6].

The requirement for unforgeability of course needs to cover naughtiness on the part of programs: they should not be able to steal capabilities from other programs, for example, nor should they be able to access an existing capability by creating a new reference to it.

I hope, by this point, you can see that capabilities could, indeed, form the basis for a very fine-grained security system. But what do they have to do with mashups and gadgets? And how on Earth do you manage all these capabilities?

# 3. MANAGING CAPABILITIES

Let us answer the second question first. The first thing to understand is that capability security has nothing to say, of itself, about how you create and who you give capabilities – other than that both these operations should be deliberate.

So, capability security does not inherently dictate any particular way of managing capabilities – it is up to the designer to choose an appropriate mechanism.

In the following sections, I describe a few ways to manage capabilities. In practice, capability systems tend to use some combination of these and other ideas.

## 3.1 Designation is Authorisation

In an interactive environment, it may often be possible to delegate the handing out of capabilities to the user himself. For example, imagine a word processing program – it needs to read the file the user wants to edit. How does it get the capability to that file?

One answer is to have the system in possession of all-powerful capabilities: for example, the capability to manufacture a read or write capability for any of the user's files. The system has a component which can interact with the user to

[5]That is, which program.

[6]Obviously, or you would be able to get hold of my bank account from the capability allowing you to put money into it.

choose a file (in other words, a file open dialog). The system endows the word processor with a capability to invoke this trusted component. When the user chooses "File/Open" the word processor invokes the file open dialog, the user chooses the file they wish to edit, the dialog returns a read capability to the word processor which can then display the contents to the user.

Contrast this with the situatation when using ACBR – from the user's point of view everything looks just the same: they choose File/Open, a dialog comes up, they choose a file, the word processor displays the fie. But under the hood something quite different has happened. The dialog returned the name of the file, the word processor asked the OS to open it, the OS checked whether the user was allowed to, and if so, opened it.

But suppose the word processor had ignored what the user chose and just used a completely different file? Of course, this would work just fine under ACBR. Indeed, the word processor wouldn't have to ask the user to choose a file at all.

But in a capability system the situation is entirely different - without the user's intervention, the word processor cannot read any files at all. Even with the user's help it can read just the file the user chose and no other.

A similar example comes from the Unix shell. Consider the standard Unix `cat` command – it takes an argument which is the name of a file to read. In a capability system, this would be translated into a capability to read just that one file. The shell would do this on behalf of the user because the user had designated that file by typing its name. Of course, the shell would have to have access to the corresponding capability (or a factory that could make it) – which would correspond the the ACBR notion of the user having permission to read the file. No further checking of permissions would be required.

## 3.2 Some Combinations are Inherently Safe

A completely different approach is also possible, and has been pioneered by Bitfrost[10][7], the security system for the XO[8]. In this the idea is that some operations are safe if they are not combined with others.

For example, it is safe to read files if you cannot communicate on the network. Conversely it is safe to communicate on the network if you cannot read local files. The threat model here is that the user's privacy might be compromised if both operations are allowed, but either one in isolation cannot expose their secrets.

In the Bitfrost model a program can ask for a set of capabilities and if they are a "safe" combination the system will grant them without user intervention. Alternatively, the user can be presented with a set of mutually exclusive safe options (e.g. "allow file access" vs. "allow network access") and can choose the appropriate one for the task at hand.

[7]Bitfrost is not actually a capability system, though it has many features inspired by capability systems.

[8]The laptop produced by the One Laptop Per Child project[4]

Although Bitfrost is the more widely known implementation, this approach was actually pioneered by the experimental CapDesk[16] system.

## 3.3 Monitors

An idea related to the Bitfrost approach is to use a system monitor – rather than granting safe combinations of capabilities in advance, instead they are granted (or revoked) according to what the program has done so far. So, for example, a program might start up being allowed to access both the filesystem and the network, but once it has done one, it is then forbidden from doing the other.

This can also be used to enforce Chinese Walls – once the user has accessed files relating to client A, they can no longer access files relating to client B, and vice versa.

The best known example of this kind of system is SE-Linux[6].

## 3.4 Extreme Limitation

This could be thought of as a subset of "some combinations are inherently safe", where the number of safe combinations is one.

In some environments it makes sense to limit what the untrusted code can do to a very small set of capabilities. For example, web gadgets (a.k.a. widgets) could be limited to

- Writing a safe subset of HTML to a designated part of the web page displayed in a browser.
- Receiving user input through forms on the web page.
- Receiving mouse clicks from the user when they are within the bounds of the part of the page allocated to the gadget.
- Making a network connection to their originating site.

This would provide useful functionality whilst effectively insulating the user from any bad things the gadget could do (such as, for example, navigating to a phishing site, or attempting to steal the user's login cookies for the embedding page).

Google's Caja project[1] can be used to do exactly this, amongst other things.

## 3.5 Access Control Lists

Many people like to think that capabilities and ACLs are in opposition – but this is not entirely so. ACLs can be used to decide whether to grant capabilities or not. In practice, this would mean that either the system would give all capabilities a user has to each program running on their behalf, or it would be prepared to hand them out on demand as the program requested them (the former technique would look more like a traditional capability-based programming model and the latter more like an ACBR one).

Note that this technique gives no advantage, of itself, over standard ACBR, unless the "unit of mistrust", so to speak, is smaller than the program itself. For example, if the program

uses a capability-secure language, then it could segregate the use of capabilities internally – so even though the program as a whole has access to all of the capabilities, each module only has those it needs to do its job. This may help when an attacker figures out how to make the program behave in unintended ways – in a pure ACBR system, the subverted module would have access to all of the permissions the whole program had access to. In a well-designed capability version of the program, the module would only have access to a small subset of those permissions (in the form of capabilities, of course).

It is also important to remember that ACLs are applied at the time the right is exercised, whereas once a capability is handed out it does not generally change, so the simplest implementation of an ACL controlled capability system is not equivalent the direct use of ACLs – if the ACLs change, the capability system may not reflect those changes.

### 3.6 Authentication and Restricting Delegation

In the purest form of capabilities, all that is required to exercise a capability is its possession. This means that, in general, it is entirely possible for the possessor of a capability to delegate it by handing a copy to someone else.

This gives rise to the capability community's fondness for the Granovetter diagram, which shows a capability for some action being handed from one party to another down an existing capability[9] (remember that in a capability system two entities can only communicate if they have a capability allowing them to do so).

However, it is sometimes desirable to authenticate the wielder of the capability, rather than relying on mere possession. There are two reasons you might want to do this

- To restrict delegation of capabilities.
- To avoid having to keep capabilities secret, where secrecy is a necessary prerequisite to unforgeability.

Note, however, that it is almost universally impossible to prevent delegation if the possessor of a capability is determined to delegate. So long as they have any kind of communication channel with the delegee, they can, at the least, exercise the capability by proxy on behalf of the delegee. And, of course, it turns out to be very hard indeed to eliminate communications channels between entities, because of covert channels.

## 4. IMPLEMENTING CAPABILITIES

So now that we know what a capability is, how do we go about creating them? There appear to be three fundamental approaches – and one could argue that these approaches derive from a choice about where to draw the boundaries between different systems.

[9]Actually Granovetter was a sociologist, and his idea was that you can only know someone if introduced by someone else. This idea is flawed, it seems to me – I can meet someone by just walking up to them in the street and saying "hi".

### 4.1 Modules

In the first, we choose to draw the boundary between individual code units (for example, objects or functions). We create a programming language that enforces capability discipline on programs written in that language.

What would such a language look like? At first, it seemed it would have to be quite esoteric, but as our understanding of capability languages has improved, we realise that capability languages are really only slightly different from standard object-oriented languages. Typically all they change is some details: no globals[10], opacity and unforgeablity of objects (which then become the capabilities themselves), access to anything outside the program itself mediated through capabilities[11].

This means we have an interesting design choice available: rather than inventing a new capability language, we can modify an existing one. The former choice is seen in E[15] and the latter in Squeak-E (Smalltalk), Joe-E[11] (Java) and Caja[1] (Javascript).

Because of the natural correspondence, mentioned above, between objects and capabilities, the concern often expressed over managing the hundreds or thousands of capabilities a fully-featured program might need is dealt with quite simply: programmers deal with thousands of objects quite easily and naturally – the fact that these objects are now capabilities does not change the ease with which they can be handled.

In this case, because capabilities are directly controlled by the language, authentication should not be required to preserve unforgeability, but may be in order to restrict delegation, depending on implementation[12].

### 4.2 Processes

The second approach is to draw the line between processes. In this case the capabilities are typically managed by the operating system, though it is possible to imagine them being managed by a privileged process.

Examples of the operating system approach are Keykos[9], Amoeba[17], EROS[14] and Coyotos[13]. Rather then access to the "rest of the world" being mediated by capability objects, such access is directly through a capability. That is, in such systems, capabilities are natively supported by the system.

Plash[5] and userv[7] could be considered to be examples of the second type of system, where capabilities are managed by a privileged process.

[10]Globals allow communication between modules, and so can bypass controls the environment is trying to impose on who gets to use what capability. Strictly speaking, globals whose transitive closure is immutable, at least by untrusted code, are ok.

[11]Of course, capabilities can also be used to mediate access to resources within the program, but this is not an inherent requirement of a capability language.

[12]I take authentication in this context to mean that the module exercising the capability is checked for the right to do so.

Once more authentication should not be needed in these systems when capabilities are controlled by the operating system, but could be needed in the case where capabilities are managed instead by a privileged process.

### 4.3 Machines

The third line we can draw is between machines. Capabilities are then manifested as network objects. One simple way to do this is to have each capability correspond to a URL whose path is a large random number[13] – this gives the required unforgeability property, so long as the URL is kept secret. Waterken[8] is an example of this, as is the E programming language, in which local objects can actually be references to objects on remote systems.

If keeping the URL secret is not possible (or desirable), then combining presentation of the URL with authentication would restore unforgeability[14]. HP's eSpeak[2] system, which combined capabilities with public keys – in order to exercise a capability you had to prove both possession of the corresponding private key and of the capability itself – is an example of this kind of system.

This mechanism is also often seen used in an ad-hoc way – for example, the confirmation mails that Mailman[3] (and other list managers) sends are effectively capabilities. Likewise, a common way to prevent cross-site request forgery[15] is to include a field in the form with a random number in it that the server can check. When an attacker attempts to forge the form submission this number will be missing or incorrect and so the attempt will fail. This number is a capability.

## 5. CAPABILITIES AND THE BROWSER

The browser is an environment that is hard to imagine how to control with ACLs, but capabilities seem to fit right in. Let's consider a gadget, for example.

A gadget is, when you get down to it, a piece of javascript supplied by one site running in a page supplied by another. From a security point of view this presents an interesting dilemma: it is very likely that the user has different levels of trust for the two pieces of code (say, for example, that the enclosing page is Google Mail and the gadget is provided by god-knows-who) – but from a traditional security point of view they are indistinguishable – they both run as the same user and they are both effectively on the same page. Furthermore the objects that one might want to protect (contact lists, contents of emails and so forth) are effectively invisible to the operating system's access control mechanisms, and to the browser's (if only it had any).

The view in a capability world could not be more different. In this case the gadget is entirely at the mercy of the enclosing page, which can decide in infinite detail what the gadget has access to and how. What's more, providing these detailed capabilities to the gadget is as easy and natural as providing Javascript objects to it. Indeed, in the case of Caja, at least, that is precisely how capabilities are implemented: as Javascript objects.

[13]Often known as a Swiss Number.

[14]Note that it is not necessary for the server to keep any kind of state in order to perform the authentication - that could be bundled into the URL itself, for example by including a public key. Obviously the URL would have to be signed in some way so that the key is bound to the capability.

[15]The attacker persuades the victim to click on a link that is equivalent to the victim submitting a form that causes some action to be performed on the victim's behalf – such as sending the attacker some money.

## 6. ADVANCED USES

It has long been held that anything capabilities can do ACLs can also do. This is not the case, and the most obvious counter example is this: if Alice wants to give Bob access to some file, say, then in an ACL system all Alice needs to do is add Bob to the ACL for that file. ACLs cannot prevent Alice from giving access to Bob.

In a capability system, Alice also needs a capability giving access to Bob in order to pass him the capability to the file[12]. Furthermore, it must be a capability whose API allows the passing of other capabilities.

The ramifications of this difference could form the subject matter of a whole book, but I give some examples here.

### 6.1 Allowing Access to Dangerous Operations

Again, thinking about the browser case; in order to render untrusted code safe it must have its output restricted so that it cannot place "dangerous" HTML on the web page (for example, `<script>` tags with arbitrary Javascript). But it may be that the container wants to allow it to use such HTML that has been "blessed" by the container.

In this case, a capability can be used to wrap the blessed HTML. The capability prevents the untrusted code from modifying its contents (by not providing a method to do so), but when handed to the safe-HTML-writing capability it bypasses the HTML safety checks and allows the blessed HTML to be written.

### 6.2 Oblivious Transport

Suppose Alice wants to tell Bob a secret, but has to do so via an intermediary, Carol. Alice can hand Carol a capability containing the secret, which has a method allowing access to the secret, but only if that method is also handed a second capability. Only Bob and Alice have this second capability. Alternatively, the second capability can have a method which can unseal the first one.

Carol can then hand that capability on to Bob, who can then combine it with the "unsealing" capability to access the data inside.

This may seem like an artificial construction, but consider the case where Alice and Bob are components of a trusted system, and Carol is untrusted code running in that system. Combining this idea with the example above, the capability handed to Carol could also contain data which, when handed to the trusted HTML renderer, would be made visible to the user, but which Carol could not herself see. Carol may determine from user actions that this capability should be used to perform some action on behalf of the user and hand it on to Bob to do so.

## 6.3 Proving User Choice

If we don't even trust Carol to correctly report the user's choices in the example above, we could elaborate this further by having the trusted renderer hand Carol a capability which proves that the user did indeed click on the capability she is now handing to Bob, and Bob could decline to act unless he sees that capability.

## 6.4 Limited Spend

Suppose I have a capability allowing unfettered access to my bank account. I could hand a capability to Alice, who I partially trust, that could be used to spend up to some limit directly from my bank account. Alice could then go and buy something on my behalf using that capability ... or steal the money from me. But Alice could not steal or spend any more than the limit I had set. Furthermore, until Alice does actually spend the money, it remains available to me in my bank account, so this is not the same as actually giving the money to her.

Also, if I thought I might change my mind later, I could first wrap the bank account capability in a revocable capability, and then wrap the revocable capability in the limited-spend capability. Note that the limited-spend capability would not need to know whether the account-access capability was revocable or not.

# 7. CONCLUSION

Identity-based access controls, and related schemes, are too coarse-grained for the requirements of modern collaborative systems. Capabilities offer the hope of fixing what otherwise appears to be a hopeless problem.

Furthermore, capabilities make some security options quite simple that would otherwise be hopelessly unwieldy.

For those interested in further exploration and experimentation, I would skip the operating system approach.

The language-based approach offers three viable alternatives, Caja, which should be in wide use by the time this paper is published, E, which is mature and functional, but not very much used, and Joe-E. Caja and Joe-E share the advantage that they are based on existing languages (Javascript and Java respectively) and so do not present a steep learning curve. E, on the other hand, has a number of interesting features, such as built-in support for writing distributed systems and an interesting and useful distributed message ordering paradigm.

For distributed systems, Waterken implements a web-based approach, and, as mentioned above, E has support at the language level.

In all cases, it makes sense to exploit the natural link between capabilities and objects by using a language designed to handle capabilities inherently, thus reducing the cognitive load on programmers and the verbosity of the code.

# 8. ACKNOWLEDGEMENTS

George Danezis, Mark Miller, Eric Sachs, Jasvir Nagra, Niels Provos, Mike Samuel, Ihab Awad, Damien Miller and Cat Okita provided valuable feedback on earlier versions of this paper.

# 9. REFERENCES

[1] Caja. Website: `http://code.google.com/p/google-caja/`.

[2] espeak. Website: `http://www.hpl.hp.co.uk/personal/Alan_Karp/espeak/`.

[3] Mailman. Website: `http://www.gnu.org/software/mailman/index.html`.

[4] one laptop per child. Website: `http://laptop.org/`.

[5] Plash: Principle of least authority shell. Website: `http://plash.beasts.org/wiki/`.

[6] Se-linux. http://www.nsa.gov/selinux/.

[7] userv. http://www.chiark.greenend.org.uk/~ian/userv/.

[8] Waterken. Website: `httpp://www.waterken.com/`.

[9] A. Bomberger, W. Frantz, A. Hardy, N. Hardy, C. Landau, and J. Shapiro. The keykos(r) nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April,*.

[10] Ivan Krstic and Simson L. Garfinkel. Bitfrost: the One Laptop per Child Security Model. *Symposium On Usable Privacy and Security*, 2007.

[11] Adrian Matthew Mettler and David Wagner. The Joe-E Language Specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 17 2006.

[12] Mark S. Miller, Ka-Ping Yee, and Jonathan S. Shapiro. Capability Myths Demolished. Technical Report Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, mar 2003.

[13] Jonathan S. Shapiro, M. Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark S. Miller. Towards a Verified, General-Purpose Operating System Kernel. In G. Klein, editor, *Proc. NICTA Invitational Workshop on Operating System Verification*, pages 1–19, National ICT Australia, 2004.

[14] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *SOSP '99: Proc. Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, New York, NY, USA, 1999. ACM Press.

[15] Marc Stiegler. The E Language in a Walnut. `http://skyhunter.com/marcs/ewalnut.html`, 2004.

[16] Marc Stiegler and Mark S. Miller. A Capability Based Client: The DarpaBrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002.

[17] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proc. 6th International Symposium on Distributed Computing Systems*, pages 558–563. IEEE, 1986.